月月android总结

### 1.AsyncTask源码分析

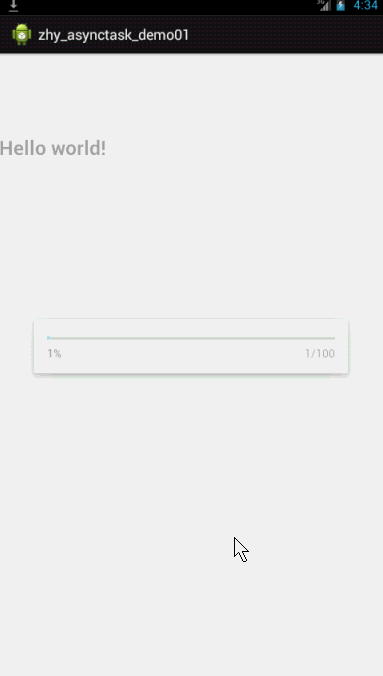
相信大家都写过这样的代码：

[java] view plain copy

1. package com.example.zhy\_asynctask\_demo01;
3. import android.app.Activity;
4. import android.app.ProgressDialog;
5. import android.os.AsyncTask;
6. import android.os.Bundle;
7. import android.util.Log;
8. import android.widget.TextView;
10. public class MainActivity extends Activity
11. {
13. private static final String TAG = "MainActivity";
14. private ProgressDialog mDialog;
15. private TextView mTextView;
17. @Override
18. protected void onCreate(Bundle savedInstanceState)
19. {
20. super.onCreate(savedInstanceState);
21. setContentView(R.layout.activity\_main);
23. mTextView = (TextView) findViewById(R.id.id\_tv);
25. mDialog = new ProgressDialog(this);
26. mDialog.setMax(100);
27. mDialog.setProgressStyle(ProgressDialog.STYLE\_HORIZONTAL);
28. mDialog.setCancelable(false);
30. new MyAsyncTask().execute();
32. }
34. private class MyAsyncTask extends AsyncTask<Void, Integer, Void>
35. {
37. @Override
38. protected void onPreExecute()
39. {
40. mDialog.show();
41. Log.e(TAG, Thread.currentThread().getName() + " onPreExecute ");
42. }
44. @Override
45. protected Void doInBackground(Void... params)
46. {
48. // 模拟数据的加载,耗时的任务
49. for (int i = 0; i < 100; i++)
50. {
51. try
52. {
53. Thread.sleep(80);
54. } catch (InterruptedException e)
55. {
56. e.printStackTrace();
57. }
58. publishProgress(i);
59. }
61. Log.e(TAG, Thread.currentThread().getName() + " doInBackground ");
62. return null;
63. }
65. @Override
66. protected void onProgressUpdate(Integer... values)
67. {
68. mDialog.setProgress(values[0]);
69. Log.e(TAG, Thread.currentThread().getName() + " onProgressUpdate ");
70. }
72. @Override
73. protected void onPostExecute(Void result)
74. {
75. // 进行数据加载完成后的UI操作
76. mDialog.dismiss();
77. mTextView.setText("LOAD DATA SUCCESS ");
78. Log.e(TAG, Thread.currentThread().getName() + " onPostExecute ");
79. }
80. }
81. }

进入某个Activity，Activity中需要的数据来自于网络或者其它耗时操作，可以在AsyncTask中onPreExecute完成一些准备操作，比如上例中显示进度对话框；然后在doInBackground完成耗时操作，在进行耗时操作时还能不时的通过publishProgress给onProgressUpdate中传递参数，然后在onProgressUpdate中可以进行UI操作，比如上例更新进度条的进度；当耗时任务执行完成后，最后在onPostExecute进行设置控件数据更新UI等操作，例如隐藏进度对话框。

效果图：



3、源码解析

注：本篇源码分析基于Andorid-17，因为和3.0之前版本变动较大，有必要标出。

那么大家一定好奇，AsyncTask在[Android](http://lib.csdn.net/base/android)中是如何实现的，下面进行源码分析：从我们的执行异步任务的起点开始，进入execute方法：

[java] view plain copy

1. public final AsyncTask<Params, Progress, Result> execute(Params... params) {
2. return executeOnExecutor(sDefaultExecutor, params);
3. }
4. public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
5. Params... params) {
6. if (mStatus != Status.PENDING) {
7. switch (mStatus) {
8. case RUNNING:
9. throw new IllegalStateException("Cannot execute task:"
10. + " the task is already running.");
11. case FINISHED:
12. throw new IllegalStateException("Cannot execute task:"
13. + " the task has already been executed "
14. + "(a task can be executed only once)");
15. }
16. }
18. mStatus = Status.RUNNING;
19. onPreExecute();
21. mWorker.mParams = params;
22. exec.execute(mFuture);
24. return this;
25. }

18行：设置当前AsyncTask的状态为RUNNING，上面的switch也可以看出，每个异步任务在完成前只能执行一次。

20行：执行了onPreExecute()，当前依然在UI线程，所以我们可以在其中做一些准备工作。

22行：将我们传入的参数赋值给了mWorker.mParams

23行：exec.execute(mFuture)

相信大家对22行出现的mWorker，以及23行出现的mFuture都会有些困惑。

mWorker找到这个类：

[java] view plain copy

1. private static abstract class WorkerRunnable<Params, Result> implements Callable<Result> {
2. Params[] mParams;
3. }

可以看到是Callable的子类，且包含一个mParams用于保存我们传入的参数，下面看初始化mWorker的代码：

[java] view plain copy

1. public AsyncTask() {
2. mWorker = new WorkerRunnable<Params, Result>() {
3. public Result call() throws Exception {
4. mTaskInvoked.set(true);
6. Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);
7. //noinspection unchecked
8. return postResult(doInBackground(mParams));
9. }
10. };
11. //….
13. }

可以看到mWorker在构造方法中完成了初始化，并且因为是一个抽象类，在这里new了一个实现类，实现了call方法，call方法中设置mTaskInvoked=true，且最终调用doInBackground(mParams)方法，并返回Result值作为参数给postResult方法.可以看到我们的doInBackground出现了，下面继续看：

[java] view plain copy

1. private Result postResult(Result result) {
2. @SuppressWarnings("unchecked")
3. Message message = sHandler.obtainMessage(MESSAGE\_POST\_RESULT,
4. new AsyncTaskResult<Result>(this, result));
5. message.sendToTarget();
6. return result;
7. }

可以看到postResult中出现了我们熟悉的异步消息机制，传递了一个消息message, message.what为MESSAGE\_POST\_RESULT；message.object= new AsyncTaskResult(this,result);

[java] view plain copy

1. private static class AsyncTaskResult<Data> {
2. final AsyncTask mTask;
3. final Data[] mData;
5. AsyncTaskResult(AsyncTask task, Data... data) {
6. mTask = task;
7. mData = data;
8. }
9. }

AsyncTaskResult就是一个简单的携带参数的对象。

看到这，我相信大家肯定会想到，在某处肯定存在一个sHandler，且复写了其handleMessage方法等待消息的传入，以及消息的处理。

[java] view plain copy

1. private static final InternalHandler sHandler = new InternalHandler();
2. private static class InternalHandler extends Handler {
3. @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
4. @Override
5. public void handleMessage(Message msg) {
6. AsyncTaskResult result = (AsyncTaskResult) msg.obj;
7. switch (msg.what) {
8. case MESSAGE\_POST\_RESULT:
9. // There is only one result
10. result.mTask.finish(result.mData[0]);
11. break;
12. case MESSAGE\_POST\_PROGRESS:
13. result.mTask.onProgressUpdate(result.mData);
14. break;
15. }
16. }
17. }

哈哈，出现了我们的handleMessage，可以看到，在接收到MESSAGE\_POST\_RESULT消息时，执行了result.mTask.finish(result.mData[0]);其实就是我们的AsyncTask.this.finish(result)，于是看finish方法

[java] view plain copy

1. private void finish(Result result) {
2. if (isCancelled()) {
3. onCancelled(result);
4. } else {
5. onPostExecute(result);
6. }
7. mStatus = Status.FINISHED;
8. }

可以看到，如果我们调用了cancel()则执行onCancelled回调；正常执行的情况下调用我们的onPostExecute(result);主要这里的调用是在handler的handleMessage中，所以是在UI线程中。如果你对异步消息机制不理解请看：[Android 异步消息处理机制 让你深入理解 Looper、Handler、Message三者关系](http://blog.csdn.net/lmj623565791/article/details/38377229)

最后将状态置为FINISHED。

mWoker看完了，应该到我们的mFuture了，依然实在构造方法中完成mFuture的初始化，将mWorker作为参数，复写了其done方法。

[java] view plain copy

1. public AsyncTask() {
2. ...
3. mFuture = new FutureTask<Result>(mWorker) {
4. @Override
5. protected void done() {
6. try {
7. postResultIfNotInvoked(get());
8. } catch (InterruptedException e) {
9. android.util.Log.w(LOG\_TAG, e);
10. } catch (ExecutionException e) {
11. throw new RuntimeException("An error occured while executing doInBackground()",
12. e.getCause());
13. } catch (CancellationException e) {
14. postResultIfNotInvoked(null);
15. }
16. }
17. };
18. }

16行：任务执行结束会调用：postResultIfNotInvoked(get());get()表示获取mWorker的call的返回值，即Result.然后看postResultIfNotInvoked方法

[java] view plain copy

1. private void postResultIfNotInvoked(Result result) {
2. final boolean wasTaskInvoked = mTaskInvoked.get();
3. if (!wasTaskInvoked) {
4. postResult(result);
5. }
6. }

如果mTaskInvoked不为true，则执行postResult；但是在mWorker初始化时就已经将mTaskInvoked为true，所以一般这个postResult执行不到。

好了，到了这里，已经介绍完了execute方法中出现了mWorker和mFurture，不过这里一直是初始化这两个对象的代码，并没有真正的执行。下面我们看真正调用执行的地方。

execute方法中的：

还记得上面的execute中的23行：exec.execute(mFuture)

exec为executeOnExecutor(sDefaultExecutor, params)中的sDefaultExecutor

下面看这个sDefaultExecutor

[java] view plain copy

1. private static volatile Executor sDefaultExecutor = SERIAL\_EXECUTOR;
2. public static final Executor SERIAL\_EXECUTOR = new SerialExecutor();
3. private static class SerialExecutor implements Executor {
4. final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
5. Runnable mActive;
6. public synchronized void execute(final Runnable r) {
7. mTasks.offer(new Runnable() {
8. public void run() {
9. try {
10. r.run();
11. } finally {
12. scheduleNext();
13. }
14. }
15. });
16. if (mActive == null) {
17. scheduleNext();
18. }
19. }
20. protected synchronized void scheduleNext() {
21. if ((mActive = mTasks.poll()) != null) {
22. THREAD\_POOL\_EXECUTOR.execute(mActive);
23. }
24. }
25. }

可以看到sDefaultExecutor其实为SerialExecutor的一个实例，其内部维持一个任务队列；直接看其execute（Runnable runnable）方法，将runnable放入mTasks队尾；

16-17行：判断当前mActive是否为空，为空则调用scheduleNext方法

20行：scheduleNext，则直接取出任务队列中的队首任务，如果不为null则传入THREAD\_POOL\_EXECUTOR进行执行。

下面看THREAD\_POOL\_EXECUTOR为何方神圣：

[java] view plain copy

1. public static final Executor THREAD\_POOL\_EXECUTOR
2. =new ThreadPoolExecutor(CORE\_POOL\_SIZE, MAXIMUM\_POOL\_SIZE, KEEP\_ALIVE,
3. TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);

可以看到就是一个自己设置参数的线程池，参数为：

[java] view plain copy

1. private static final int CORE\_POOL\_SIZE = 5;
2. private static final int MAXIMUM\_POOL\_SIZE = 128;
3. private static final int KEEP\_ALIVE = 1;
4. private static final ThreadFactory sThreadFactory = new ThreadFactory() {
5. private final AtomicInteger mCount = new AtomicInteger(1);
6. public Thread newThread(Runnable r) {
7. return new Thread(r, "AsyncTask #" + mCount.getAndIncrement());
8. }
9. };
10. private static final BlockingQueue<Runnable> sPoolWorkQueue =
11. new LinkedBlockingQueue<Runnable>(10);

看到这里，大家可能会认为，背后原来有一个线程池，且最大支持128的线程并发，加上长度为10的阻塞队列，可能会觉得就是在快速调用138个以内的AsyncTask子类的execute方法不会出现问题，而大于138则会抛出异常。

其实不是这样的，我们再仔细看一下代码，回顾一下sDefaultExecutor，真正在execute()中调用的为sDefaultExecutor.execute：

[java] view plain copy

1. private static class SerialExecutor implements Executor {
2. final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
3. Runnable mActive;
4. public synchronized void execute(final Runnable r) {
5. mTasks.offer(new Runnable() {
6. public void run() {
7. try {
8. r.run();
9. } finally {
10. scheduleNext();
11. }
12. }
13. });
14. if (mActive == null) {
15. scheduleNext();
16. }
17. }
18. protected synchronized void scheduleNext() {
19. if ((mActive = mTasks.poll()) != null) {
20. THREAD\_POOL\_EXECUTOR.execute(mActive);
21. }
22. }
23. }

可以看到，如果此时有10个任务同时调用execute（s synchronized）方法，第一个任务入队，然后在mActive = mTasks.poll()) != null被取出，并且赋值给mActivte，然后交给线程池去执行。然后第二个任务入队，但是此时mActive并不为null，并不会执行scheduleNext();所以如果第一个任务比较慢，10个任务都会进入队列等待；真正执行下一个任务的时机是，线程池执行完成第一个任务以后，调用Runnable中的finally代码块中的scheduleNext，所以虽然内部有一个线程池，其实调用的过程还是线性的。一个接着一个的执行，相当于单线程。

4、总结

到此源码解释完毕，由于代码跨度比较大，我们再回顾一下：

[java] view plain copy

1. public final AsyncTask<Params, Progress, Result> execute(Params... params) {
2. return executeOnExecutor(sDefaultExecutor, params);
3. }
4. public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
5. Params... params) {
6. if (mStatus != Status.PENDING) {
7. switch (mStatus) {
8. case RUNNING:
9. throw new IllegalStateException("Cannot execute task:"
10. + " the task is already running.");
11. case FINISHED:
12. throw new IllegalStateException("Cannot execute task:"
13. + " the task has already been executed "
14. + "(a task can be executed only once)");
15. }
16. }
18. mStatus = Status.RUNNING;
20. onPreExecute();
22. mWorker.mParams = params;
23. exec.execute(mFuture);
25. return this;
26. }

18行：设置当前AsyncTask的状态为RUNNING，上面的switch也可以看出，每个异步任务在完成前只能执行一次。

20行：执行了onPreExecute()，当前依然在UI线程，所以我们可以在其中做一些准备工作。

22行：将我们传入的参数赋值给了mWorker.mParams ,mWorker为一个Callable的子类，且在内部的call()方法中，调用了doInBackground(mParams)，然后得到的返回值作为postResult的参数进行执行；postResult中通过sHandler发送消息，最终sHandler的handleMessage中完成onPostExecute的调用。

23行：exec.execute(mFuture)，mFuture为真正的执行任务的单元，将mWorker进行封装，然后由sDefaultExecutor交给线程池进行执行。

5、publishProgress

说了这么多，我们好像忘了一个方法：publishProgress

[java] view plain copy

1. protected final void publishProgress(Progress... values) {
2. if (!isCancelled()) {
3. sHandler.obtainMessage(MESSAGE\_POST\_PROGRESS,
4. new AsyncTaskResult<Progress>(this, values)).sendToTarget();
5. }
6. }

也很简单，直接使用sHandler发送一个消息，并且携带我们传入的值；

[java] view plain copy

1. private static class InternalHandler extends Handler {
2. @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
3. @Override
4. public void handleMessage(Message msg) {
5. AsyncTaskResult result = (AsyncTaskResult) msg.obj;
6. switch (msg.what) {
7. case MESSAGE\_POST\_RESULT:
8. // There is only one result
9. result.mTask.finish(result.mData[0]);
10. break;
11. case MESSAGE\_POST\_PROGRESS:
12. result.mTask.onProgressUpdate(result.mData);
13. break;
14. }
15. }
16. }

在handleMessage中进行了我们的onProgressUpdate(result.mData);的调用。

6、AsyncTask曾经缺陷

记得以前有个面试题经常会问道：AsyncTask运行的原理是什么？有什么缺陷？

以前对于缺陷的答案可能是：AsyncTask在并发执行多个任务时发生异常。其实还是存在的，在3.0以前的系统中还是会以支持多线程并发的方式执行，支持并发数也是我们上面所计算的128，阻塞队列可以存放10个；也就是同时执行138个任务是没有问题的；而超过138会马上出现[Java](http://lib.csdn.net/base/java).util.concurrent.RejectedExecutionException；

而在在3.0以上包括3.0的系统中会为单线程执行（即我们上面代码的分析）；

空说无凭：下面看[测试](http://lib.csdn.net/base/softwaretest)代码：

[java] view plain copy

1. package com.example.zhy\_asynctask\_demo01;
3. import android.app.Activity;
4. import android.app.ProgressDialog;
5. import android.os.AsyncTask;
6. import android.os.Bundle;
7. import android.util.Log;
8. import android.widget.TextView;
10. public class MainActivity extends Activity
11. {
13. private static final String TAG = "MainActivity";
14. private ProgressDialog mDialog;
15. private TextView mTextView;
17. @Override
18. protected void onCreate(Bundle savedInstanceState)
19. {
20. super.onCreate(savedInstanceState);
21. setContentView(R.layout.activity\_main);
23. mTextView = (TextView) findViewById(R.id.id\_tv);
25. mDialog = new ProgressDialog(this);
26. mDialog.setMax(100);
27. mDialog.setProgressStyle(ProgressDialog.STYLE\_HORIZONTAL);
28. mDialog.setCancelable(false);

31. for(int i = 1 ;i <= 138 ; i++)
32. {
33. new MyAsyncTask2().execute();
34. }
36. //new MyAsyncTask().execute();

39. }
41. private class MyAsyncTask2 extends AsyncTask<Void,Void, Void>
42. {
44. @Override
45. protected Void doInBackground(Void... params)
46. {
47. try
48. {
49. Log.e(TAG, Thread.currentThread().getName());
50. Thread.sleep(10000);
51. } catch (InterruptedException e)
52. {
53. e.printStackTrace();
54. }
55. return null;
56. }
58. }
59. }

可以看到我for循环中执行138个异步任务，每个异步任务的执行需要10s;下面使用2.2的模拟器进行测试：

输出结果为：

AsyncTask#1 - AsyncTask #128同时输出

然后10s后，另外10个任务输出。

可以分析结果，得到结论：AsyncTask在2.2的系统中同时支持128个任务并发，至少支持10个任务等待；

下面将138个任务，改成139个任务：

[java] view plain copy

1. for(int i = 1 ;i <= 139 ; i++)
2. {
3. new MyAsyncTask2().execute();
4. }

运行结果：会发生异常：java.util.concurrent.RejectedExecutionException ； 于是可以确定仅支持10个任务等待，超过10个则立即发生异常。

简单说一下出现异常的原因：现在是139个任务，几乎同时提交，线程池支持128个的并发，然后阻塞队列数量为10，此时当第11个任务提交的时候则会发生异常。

简单看一下源码：

[java] view plain copy

1. public static final Executor THREAD\_POOL\_EXECUTOR
2. = new ThreadPoolExecutor(CORE\_POOL\_SIZE, MAXIMUM\_POOL\_SIZE, KEEP\_ALIVE, TimeUnit.SECONDS, sPoolWorkQueue, sThreadFactory);

看ThreadPoolExecutor的execute方法：

[java] view plain copy

1. if (isRunning(c) && workQueue.offer(command)) {
2. int recheck = ctl.get();
3. if (! isRunning(recheck) && remove(command))
4. reject(command);
5. else if (workerCountOf(recheck) == 0)
6. addWorker(null, false);
7. }
8. else if (!addWorker(command, false))
9. reject(command);

当阻塞队列满的时候workQueue.offer(command)返回false;然后执行addWorker(command,false)方法，如果返回false则执行reject()方法.

[java] view plain copy

1. private boolean addWorker(Runnable firstTask, boolean core) {
2. …
3. int wc = workerCountOf(c);
4. if (wc >= CAPACITY ||
5. wc >= (core ? corePoolSize : maximumPoolSize))
6. return false;
7. …
8. }

可以看到当任务数目大于容量则返回false，最终在reject()中抛出异常。

上面就是使用2.2模拟器测试的结果；

下面将系统改为4.1.1，也就是我的测试机小米2s

把线程数改为139甚至1000，你可以看到任务一个接一个的在那缓慢的执行，不会抛什么异常，不过线程倒是1个1个的在那执行；

好了，如果现在大家去面试，被问到AsyncTask的缺陷，可以分为两个部分说，在3.0以前，最大支持128个线程的并发，10个任务的等待。在3.0以后，无论有多少任务，都会在其内部单线程执行；

至此，AsyncTask源码分析完毕，相信大家对AsyncTask有了更深的理解~~~

以上是hongyang的分析

以下是郭霖的分析

我们都知道，[**Android**](http://lib.csdn.net/base/android) UI是线程不安全的，如果想要在子线程里进行UI操作，就需要借助[**android**](http://lib.csdn.net/base/android)的异步消息处理机制。之前我也写过了一篇文章从源码层面分析了Android的异步消息处理机制，感兴趣的朋友可以参考 [**Android Handler、Message完全解析，带你从源码的角度彻底理解**](http://blog.csdn.net/guolin_blog/article/details/9991569) 。

不过为了更加方便我们在子线程中更新UI元素，Android从1.5版本就引入了一个AsyncTask类，使用它就可以非常灵活方便地从子线程切换到UI线程，我们本篇文章的主角也就正是它了。

AsyncTask很早就出现在Android的API里了，所以我相信大多数朋友对它的用法都已经非常熟悉。不过今天我还是准备从AsyncTask的基本用法开始讲起，然后我们再来一起分析下AsyncTask源码，看看它是如何实现的，最后我会介绍一些关于AsyncTask你所不知道的秘密。

**AsyncTask的基本用法**

首先来看一下AsyncTask的基本用法，由于AsyncTask是一个抽象类，所以如果我们想使用它，就必须要创建一个子类去继承它。在继承时我们可以为AsyncTask类指定三个泛型参数，这三个参数的用途如下：

1. Params

在执行AsyncTask时需要传入的参数，可用于在后台任务中使用。

2. Progress

后台任务执行时，如果需要在界面上显示当前的进度，则使用这里指定的泛型作为进度单位。

3. Result

当任务执行完毕后，如果需要对结果进行返回，则使用这里指定的泛型作为返回值类型。

因此，一个最简单的自定义AsyncTask就可以写成如下方式：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **class** DownloadTask **extends** AsyncTask<Void, Integer, Boolean> {
2. ……
3. }

这里我们把AsyncTask的第一个泛型参数指定为Void，表示在执行AsyncTask的时候不需要传入参数给后台任务。第二个泛型参数指定为Integer，表示使用整型数据来作为进度显示单位。第三个泛型参数指定为Boolean，则表示使用布尔型数据来反馈执行结果。

当然，目前我们自定义的DownloadTask还是一个空任务，并不能进行任何实际的操作，我们还需要去重写AsyncTask中的几个方法才能完成对任务的定制。经常需要去重写的方法有以下四个：

1. onPreExecute()

这个方法会在后台任务开始执行之间调用，用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

2. doInBackground(Params...)

这个方法中的所有代码都会在子线程中运行，我们应该在这里去处理所有的耗时任务。任务一旦完成就可以通过return语句来将任务的执行结果进行返回，如果AsyncTask的第三个泛型参数指定的是Void，就可以不返回任务执行结果。注意，在这个方法中是不可以进行UI操作的，如果需要更新UI元素，比如说反馈当前任务的执行进度，可以调用publishProgress(Progress...)方法来完成。

3. onProgressUpdate(Progress...)

当在后台任务中调用了publishProgress(Progress...)方法后，这个方法就很快会被调用，方法中携带的参数就是在后台任务中传递过来的。在这个方法中可以对UI进行操作，利用参数中的数值就可以对界面元素进行相应的更新。

4. onPostExecute(Result)

当后台任务执行完毕并通过return语句进行返回时，这个方法就很快会被调用。返回的数据会作为参数传递到此方法中，可以利用返回的数据来进行一些UI操作，比如说提醒任务执行的结果，以及关闭掉进度条对话框等。

因此，一个比较完整的自定义AsyncTask就可以写成如下方式：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **class** DownloadTask **extends** AsyncTask<Void, Integer, Boolean> {
3. @Override
4. **protected** **void** onPreExecute() {
5. progressDialog.show();
6. }
8. @Override
9. **protected** Boolean doInBackground(Void... params) {
10. **try** {
11. **while** (**true**) {
12. **int** downloadPercent = doDownload();
13. publishProgress(downloadPercent);
14. **if** (downloadPercent >= 100) {
15. **break**;
16. }
17. }
18. } **catch** (Exception e) {
19. **return** **false**;
20. }
21. **return** **true**;
22. }
24. @Override
25. **protected** **void** onProgressUpdate(Integer... values) {
26. progressDialog.setMessage("当前下载进度：" + values[0] + "%");
27. }
29. @Override
30. **protected** **void** onPostExecute(Boolean result) {
31. progressDialog.dismiss();
32. **if** (result) {
33. Toast.makeText(context, "下载成功", Toast.LENGTH\_SHORT).show();
34. } **else** {
35. Toast.makeText(context, "下载失败", Toast.LENGTH\_SHORT).show();
36. }
37. }
38. }

这里我们模拟了一个下载任务，在doInBackground()方法中去执行具体的下载逻辑，在onProgressUpdate()方法中显示当前的下载进度，在onPostExecute()方法中来提示任务的执行结果。如果想要启动这个任务，只需要简单地调用以下代码即可：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **new** DownloadTask().execute();

以上就是AsyncTask的基本用法，怎么样，是不是感觉在子线程和UI线程之间进行切换变得灵活了很多？我们并不需求去考虑什么异步消息处理机制，也不需要专门使用一个Handler来发送和接收消息，只需要调用一下publishProgress()方法就可以轻松地从子线程切换到UI线程了。

**分析AsyncTask的源码**

虽然AsyncTask这么简单好用，但你知道它是怎样实现的吗？那么接下来，我们就来分析一下AsyncTask的源码，对它的实现原理一探究竟。注意这里我选用的是Android 4.0的源码，如果你查看的是其它版本的源码，可能会有一些出入。

从之前DownloadTask的代码就可以看出，在启动某一个任务之前，要先new出它的实例，因此，我们就先来看一看AsyncTask构造函数中的源码，如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **public** AsyncTask() {
2. mWorker = **new** WorkerRunnable<Params, Result>() {
3. **public** Result call() **throws** Exception {
4. mTaskInvoked.set(**true**);
5. Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);
6. **return** postResult(doInBackground(mParams));
7. }
8. };
9. mFuture = **new** FutureTask<Result>(mWorker) {
10. @Override
11. **protected** **void** done() {
12. **try** {
13. **final** Result result = get();
14. postResultIfNotInvoked(result);
15. } **catch** (InterruptedException e) {
16. android.util.Log.w(LOG\_TAG, e);
17. } **catch** (ExecutionException e) {
18. **throw** **new** RuntimeException("An error occured while executing doInBackground()",
19. e.getCause());
20. } **catch** (CancellationException e) {
21. postResultIfNotInvoked(**null**);
22. } **catch** (Throwable t) {
23. **throw** **new** RuntimeException("An error occured while executing "
24. + "doInBackground()", t);
25. }
26. }
27. };
28. }

这段代码虽然看起来有点长，但实际上并没有任何具体的逻辑会得到执行，只是初始化了两个变量，mWorker和mFuture，并在初始化mFuture的时候将mWorker作为参数传入。mWorker是一个Callable对象，mFuture是一个FutureTask对象，这两个变量会暂时保存在内存中，稍后才会用到它们。

接着如果想要启动某一个任务，就需要调用该任务的execute()方法，因此现在我们来看一看execute()方法的源码，如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **public** **final** AsyncTask<Params, Progress, Result> execute(Params... params) {
2. **return** executeOnExecutor(sDefaultExecutor, params);
3. }

简单的有点过分了，只有一行代码，仅是调用了executeOnExecutor()方法，那么具体的逻辑就应该写在这个方法里了，快跟进去瞧一瞧：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **public** **final** AsyncTask<Params, Progress, Result> executeOnExecutor(Executor exec,
2. Params... params) {
3. **if** (mStatus != Status.PENDING) {
4. **switch** (mStatus) {
5. **case** RUNNING:
6. **throw** **new** IllegalStateException("Cannot execute task:"
7. + " the task is already running.");
8. **case** FINISHED:
9. **throw** **new** IllegalStateException("Cannot execute task:"
10. + " the task has already been executed "
11. + "(a task can be executed only once)");
12. }
13. }
14. mStatus = Status.RUNNING;
15. onPreExecute();
16. mWorker.mParams = params;
17. exec.execute(mFuture);
18. **return** **this**;
19. }

果然，这里的代码看上去才正常点。可以看到，在第15行调用了onPreExecute()方法，因此证明了onPreExecute()方法会第一个得到执行。可是接下来的代码就看不明白了，怎么没见到哪里有调用doInBackground()方法呢？别着急，慢慢找总会找到的，我们看到，在第17行调用了Executor的execute()方法，并将前面初始化的mFuture对象传了进去，那么这个Executor对象又是什么呢？查看上面的execute()方法，原来是传入了一个sDefaultExecutor变量，接着找一下这个sDefaultExecutor变量是在哪里定义的，源码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **public** **static** **final** Executor SERIAL\_EXECUTOR = **new** SerialExecutor();
2. ……
3. **private** **static** **volatile** Executor sDefaultExecutor = SERIAL\_EXECUTOR;

可以看到，这里先new出了一个SERIAL\_EXECUTOR常量，然后将sDefaultExecutor的值赋值为这个常量，也就是说明，刚才在executeOnExecutor()方法中调用的execute()方法，其实也就是调用的SerialExecutor类中的execute()方法。那么我们自然要去看看SerialExecutor的源码了，如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **private** **static** **class** SerialExecutor **implements** Executor {
2. **final** ArrayDeque<Runnable> mTasks = **new** ArrayDeque<Runnable>();
3. Runnable mActive;
5. **public** **synchronized** **void** execute(**final** Runnable r) {
6. mTasks.offer(**new** Runnable() {
7. **public** **void** run() {
8. **try** {
9. r.run();
10. } **finally** {
11. scheduleNext();
12. }
13. }
14. });
15. **if** (mActive == **null**) {
16. scheduleNext();
17. }
18. }
20. **protected** **synchronized** **void** scheduleNext() {
21. **if** ((mActive = mTasks.poll()) != **null**) {
22. THREAD\_POOL\_EXECUTOR.execute(mActive);
23. }
24. }
25. }

SerialExecutor类中也有一个execute()方法，这个方法里的所有逻辑就是在子线程中执行的了，注意这个方法有一个Runnable参数，那么目前这个参数的值是什么呢？当然就是mFuture对象了，也就是说在第9行我们要调用的是FutureTask类的run()方法，而在这个方法里又会去调用Sync内部类的innerRun()方法，因此我们直接来看innerRun()方法的源码：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **void** innerRun() {
2. **if** (!compareAndSetState(READY, RUNNING))
3. **return**;
4. runner = Thread.currentThread();
5. **if** (getState() == RUNNING) { // recheck after setting thread
6. V result;
7. **try** {
8. result = callable.call();
9. } **catch** (Throwable ex) {
10. setException(ex);
11. **return**;
12. }
13. set(result);
14. } **else** {
15. releaseShared(0); // cancel
16. }
17. }

可以看到，在第8行调用了callable的call()方法，那么这个callable对象是什么呢？其实就是在初始化mFuture对象时传入的mWorker对象了，此时调用的call()方法，也就是一开始在AsyncTask的构造函数中指定的，我们把它单独拿出来看一下，代码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **public** Result call() **throws** Exception {
2. mTaskInvoked.set(**true**);
3. Process.setThreadPriority(Process.THREAD\_PRIORITY\_BACKGROUND);
4. **return** postResult(doInBackground(mParams));
5. }

在postResult()方法的参数里面，我们终于找到了doInBackground()方法的调用处，虽然经过了很多周转，但目前的代码仍然是运行在子线程当中的，所以这也就是为什么我们可以在doInBackground()方法中去处理耗时的逻辑。接着将doInBackground()方法返回的结果传递给了postResult()方法，这个方法的源码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **private** Result postResult(Result result) {
2. Message message = sHandler.obtainMessage(MESSAGE\_POST\_RESULT,
3. **new** AsyncTaskResult<Result>(**this**, result));
4. message.sendToTarget();
5. **return** result;
6. }

如果你已经熟悉了异步消息处理机制，这段代码对你来说一定非常简单吧。这里使用sHandler对象发出了一条消息，消息中携带了MESSAGE\_POST\_RESULT常量和一个表示任务执行结果的AsyncTaskResult对象。这个sHandler对象是InternalHandler类的一个实例，那么稍后这条消息肯定会在InternalHandler的handleMessage()方法中被处理。InternalHandler的源码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **private** **static** **class** InternalHandler **extends** Handler {
2. @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
3. @Override
4. **public** **void** handleMessage(Message msg) {
5. AsyncTaskResult result = (AsyncTaskResult) msg.obj;
6. **switch** (msg.what) {
7. **case** MESSAGE\_POST\_RESULT:
8. // There is only one result
9. result.mTask.finish(result.mData[0]);
10. **break**;
11. **case** MESSAGE\_POST\_PROGRESS:
12. result.mTask.onProgressUpdate(result.mData);
13. **break**;
14. }
15. }
16. }

这里对消息的类型进行了判断，如果这是一条MESSAGE\_POST\_RESULT消息，就会去执行finish()方法，如果这是一条MESSAGE\_POST\_PROGRESS消息，就会去执行onProgressUpdate()方法。那么finish()方法的源码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **private** **void** finish(Result result) {
2. **if** (isCancelled()) {
3. onCancelled(result);
4. } **else** {
5. onPostExecute(result);
6. }
7. mStatus = Status.FINISHED;
8. }

可以看到，如果当前任务被取消掉了，就会调用onCancelled()方法，如果没有被取消，则调用onPostExecute()方法，这样当前任务的执行就全部结束了。

我们注意到，在刚才InternalHandler的handleMessage()方法里，还有一种MESSAGE\_POST\_PROGRESS的消息类型，这种消息是用于当前进度的，调用的正是onProgressUpdate()方法，那么什么时候才会发出这样一条消息呢？相信你已经猜到了，查看publishProgress()方法的源码，如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **protected** **final** **void** publishProgress(Progress... values) {
2. **if** (!isCancelled()) {
3. sHandler.obtainMessage(MESSAGE\_POST\_PROGRESS,
4. **new** AsyncTaskResult<Progress>(**this**, values)).sendToTarget();
5. }
6. }

非常清晰了吧！正因如此，在doInBackground()方法中调用publishProgress()方法才可以从子线程切换到UI线程，从而完成对UI元素的更新操作。其实也没有什么神秘的，因为说到底，AsyncTask也是使用的异步消息处理机制，只是做了非常好的封装而已。

读到这里，相信你对AsyncTask中的每个回调方法的作用、原理、以及何时会被调用都已经搞明白了吧。

**关于AsyncTask你所不知道的秘密**

不得不说，刚才我们在分析SerialExecutor的时候，其实并没有分析的很仔细，仅仅只是关注了它会调用mFuture中的run()方法，但是至于什么时候会调用我们并没有进一步地研究。其实SerialExecutor也是AsyncTask在3.0版本以后做了最主要的修改的地方，它在AsyncTask中是以常量的形式被使用的，因此在整个应用程序中的所有AsyncTask实例都会共用同一个SerialExecutor。下面我们就来对这个类进行更加详细的分析，为了方便阅读，我把它的代码再贴出来一遍：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **private** **static** **class** SerialExecutor **implements** Executor {
2. **final** ArrayDeque<Runnable> mTasks = **new** ArrayDeque<Runnable>();
3. Runnable mActive;
5. **public** **synchronized** **void** execute(**final** Runnable r) {
6. mTasks.offer(**new** Runnable() {
7. **public** **void** run() {
8. **try** {
9. r.run();
10. } **finally** {
11. scheduleNext();
12. }
13. }
14. });
15. **if** (mActive == **null**) {
16. scheduleNext();
17. }
18. }
20. **protected** **synchronized** **void** scheduleNext() {
21. **if** ((mActive = mTasks.poll()) != **null**) {
22. THREAD\_POOL\_EXECUTOR.execute(mActive);
23. }
24. }
25. }

可以看到，SerialExecutor是使用ArrayDeque这个队列来管理Runnable对象的，如果我们一次性启动了很多个任务，首先在第一次运行execute()方法的时候，会调用ArrayDeque的offer()方法将传入的Runnable对象添加到队列的尾部，然后判断mActive对象是不是等于null，第一次运行当然是等于null了，于是会调用scheduleNext()方法。在这个方法中会从队列的头部取值，并赋值给mActive对象，然后调用THREAD\_POOL\_EXECUTOR去执行取出的取出的Runnable对象。之后如何又有新的任务被执行，同样还会调用offer()方法将传入的Runnable添加到队列的尾部，但是再去给mActive对象做非空检查的时候就会发现mActive对象已经不再是null了，于是就不会再调用scheduleNext()方法。

那么后面添加的任务岂不是永远得不到处理了？当然不是，看一看offer()方法里传入的Runnable匿名类，这里使用了一个try finally代码块，并在finally中调用了scheduleNext()方法，保证无论发生什么情况，这个方法都会被调用。也就是说，每次当一个任务执行完毕后，下一个任务才会得到执行，SerialExecutor模仿的是单一线程池的效果，如果我们快速地启动了很多任务，同一时刻只会有一个线程正在执行，其余的均处于等待状态。[**Android照片墙应用实现，再多的图片也不怕崩溃**](http://blog.csdn.net/guolin_blog/article/details/9526203) 这篇文章中例子的运行结果也证实了这个结论。

不过你可能还不知道，在Android 3.0之前是并没有SerialExecutor这个类的，那个时候是直接在AsyncTask中构建了一个sExecutor常量，并对线程池总大小，同一时刻能够运行的线程数做了规定，代码如下所示：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. **private** **static** **final** **int** CORE\_POOL\_SIZE = 5;
2. **private** **static** **final** **int** MAXIMUM\_POOL\_SIZE = 128;
3. **private** **static** **final** **int** KEEP\_ALIVE = 10;
4. ……
5. **private** **static** **final** ThreadPoolExecutor sExecutor = **new** ThreadPoolExecutor(CORE\_POOL\_SIZE,
6. MAXIMUM\_POOL\_SIZE, KEEP\_ALIVE, TimeUnit.SECONDS, sWorkQueue, sThreadFactory);

可以看到，这里规定同一时刻能够运行的线程数为5个，线程池总大小为128。也就是说当我们启动了10个任务时，只有5个任务能够立刻执行，另外的5个任务则需要等待，当有一个任务执行完毕后，第6个任务才会启动，以此类推。而线程池中最大能存放的线程数是128个，当我们尝试去添加第129个任务时，程序就会崩溃。

因此在3.0版本中AsyncTask的改动还是挺大的，在3.0之前的AsyncTask可以同时有5个任务在执行，而3.0之后的AsyncTask同时只能有1个任务在执行。为什么升级之后可以同时执行的任务数反而变少了呢？这是因为更新后的AsyncTask已变得更加灵活，如果不想使用默认的线程池，还可以自由地进行配置。比如使用如下的代码来启动任务：

**[java]** [view plain](http://blog.csdn.net/guolin_blog/article/details/11711405) [copy](http://blog.csdn.net/guolin_blog/article/details/11711405)

1. Executor exec = **new** ThreadPoolExecutor(15, 200, 10,
2. TimeUnit.SECONDS, **new** LinkedBlockingQueue<Runnable>());
3. **new** DownloadTask().executeOnExecutor(exec);

这样就可以使用我们自定义的一个Executor来执行任务，而不是使用SerialExecutor。上述代码的效果允许在同一时刻有15个任务正在执行，并且最多能够存储200个任务。

好了，到这里我们就已经把关于AsyncTask的所有重要内容深入浅出地理解了一遍，相信在将来使用它的时候能够更加得心应手。

### 2．[Android事件分发机制完全解析，带你从源码的角度彻底理解(上)](http://blog.csdn.net/guolin_blog/article/details/9097463)

阅读源码讲究由浅入深，循序渐进，因此我们也从简单的开始，本篇先带大家探究View的事件分发，下篇再去探究难度更高的ViewGroup的事件分发。

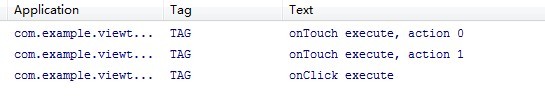
那我们现在就开始吧！比如说你当前有一个非常简单的项目，只有一个Activity，并且Activity中只有一个按钮。你可能已经知道，如果想要给这个按钮注册一个点击事件，只需要调用：

1. button.setOnClickListener(**new** OnClickListener() {
2. @Override
3. **public** **void** onClick(View v) {
4. Log.d("TAG", "onClick execute");
5. }
6. });

这样在onClick方法里面写实现，就可以在按钮被点击的时候执行。你可能也已经知道，如果想给这个按钮再添加一个touch事件，只需要调用：

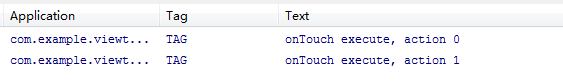
1. button.setOnTouchListener(**new** OnTouchListener() {
2. @Override
3. **public** **boolean** onTouch(View v, MotionEvent event) {
4. Log.d("TAG", "onTouch execute, action " + event.getAction());
5. **return** **false**;
6. }
7. });

onTouch方法里能做的事情比onClick要多一些，比如判断手指按下、抬起、移动等事件。那么如果我两个事件都注册了，哪一个会先执行呢？我们来试一下就知道了，运行程序点击按钮，打印结果如下：



可以看到，onTouch是优先于onClick执行的，并且onTouch执行了两次，一次是ACTION\_DOWN，一次是ACTION\_UP(你还可能会有多次ACTION\_MOVE的执行，如果你手抖了一下)。因此事件传递的顺序是先经过onTouch，再传递到onClick。

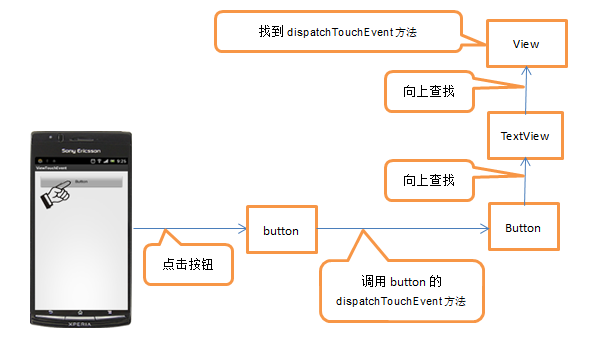
细心的朋友应该可以注意到，onTouch方法是有返回值的，这里我们返回的是false，如果我们尝试把onTouch方法里的返回值改成true，再运行一次，结果如下：



我们发现，onClick方法不再执行了！为什么会这样呢？你可以先理解成onTouch方法返回true就认为这个事件被onTouch消费掉了，因而不会再继续向下传递。

如果到现在为止，以上的所有知识点你都是清楚的，那么说明你对Android事件传递的基本用法应该是掌握了。不过别满足于现状，让我们从源码的角度分析一下，出现上述现象的原理是什么。

首先你需要知道一点，只要你触摸到了任何一个控件，就一定会调用该控件的dispatchTouchEvent方法。那当我们去点击按钮的时候，就会去调用Button类里的dispatchTouchEvent方法，可是你会发现Button类里并没有这个方法，那么就到它的父类TextView里去找一找，你会发现TextView里也没有这个方法，那没办法了，只好继续在TextView的父类View里找一找，这个时候你终于在View里找到了这个方法，示意图如下：



然后我们来看一下View中dispatchTouchEvent方法的源码：

**[java]** view plain copy

1. **public** **boolean** dispatchTouchEvent(MotionEvent event) {
2. **if** (mOnTouchListener != **null** && (mViewFlags & ENABLED\_MASK) == ENABLED &&
3. mOnTouchListener.onTouch(**this**, event)) {
4. **return** **true**;
5. }
6. **return** onTouchEvent(event);
7. }

这个方法非常的简洁，只有短短几行代码！我们可以看到，在这个方法内，首先是进行了一个判断，如果mOnTouchListener != null，(mViewFlags & ENABLED\_MASK) == ENABLED和mOnTouchListener.onTouch(this, event)这三个条件都为真，就返回true，否则就去执行onTouchEvent(event)方法并返回。

先看一下第一个条件，mOnTouchListener这个变量是在哪里赋值的呢？我们寻找之后在View里发现了如下方法：

**[java]** view plain copy

1. **public** **void** setOnTouchListener(OnTouchListener l) {
2. mOnTouchListener = l;
3. }

Bingo！找到了，mOnTouchListener正是在setOnTouchListener方法里赋值的，也就是说只要我们给控件注册了touch事件，mOnTouchListener就一定被赋值了。

第二个条件(mViewFlags & ENABLED\_MASK) == ENABLED是判断当前点击的控件是否是enable的，按钮默认都是enable的，因此这个条件恒定为true。

第三个条件就比较关键了，mOnTouchListener.onTouch(this, event)，其实也就是去回调控件注册touch事件时的onTouch方法。也就是说如果我们在onTouch方法里返回true，就会让这三个条件全部成立，从而整个方法直接返回true。如果我们在onTouch方法里返回false，就会再去执行onTouchEvent(event)方法。

现在我们可以结合前面的例子来分析一下了，首先在dispatchTouchEvent中最先执行的就是onTouch方法，因此onTouch肯定是要优先于onClick执行的，也是印证了刚刚的打印结果。而如果在onTouch方法里返回了true，就会让dispatchTouchEvent方法直接返回true，不会再继续往下执行。而打印结果也证实了如果onTouch返回true，onClick就不会再执行了。

根据以上源码的分析，从原理上解释了我们前面例子的运行结果。而上面的分析还透漏出了一个重要的信息，那就是onClick的调用肯定是在onTouchEvent(event)方法中的！那我们马上来看下onTouchEvent的源码，如下所示：

**[java]** view plain copy

1. **public** **boolean** onTouchEvent(MotionEvent event) {
2. **final** **int** viewFlags = mViewFlags;
3. **if** ((viewFlags & ENABLED\_MASK) == DISABLED) {
4. // A disabled view that is clickable still consumes the touch
5. // events, it just doesn't respond to them.
6. **return** (((viewFlags & CLICKABLE) == CLICKABLE ||
7. (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE));
8. }
9. **if** (mTouchDelegate != **null**) {
10. **if** (mTouchDelegate.onTouchEvent(event)) {
11. **return** **true**;
12. }
13. }
14. **if** (((viewFlags & CLICKABLE) == CLICKABLE ||
15. (viewFlags & LONG\_CLICKABLE) == LONG\_CLICKABLE)) {
16. **switch** (event.getAction()) {
17. **case** MotionEvent.ACTION\_UP:
18. **boolean** prepressed = (mPrivateFlags & PREPRESSED) != 0;
19. **if** ((mPrivateFlags & PRESSED) != 0 || prepressed) {
20. // take focus if we don't have it already and we should in
21. // touch mode.
22. **boolean** focusTaken = **false**;
23. **if** (isFocusable() && isFocusableInTouchMode() && !isFocused()) {
24. focusTaken = requestFocus();
25. }
26. **if** (!mHasPerformedLongPress) {
27. // This is a tap, so remove the longpress check
28. removeLongPressCallback();
29. // Only perform take click actions if we were in the pressed state
30. **if** (!focusTaken) {
31. // Use a Runnable and post this rather than calling
32. // performClick directly. This lets other visual state
33. // of the view update before click actions start.
34. **if** (mPerformClick == **null**) {
35. mPerformClick = **new** PerformClick();
36. }
37. **if** (!post(mPerformClick)) {
38. performClick();
39. }
40. }
41. }
42. **if** (mUnsetPressedState == **null**) {
43. mUnsetPressedState = **new** UnsetPressedState();
44. }
45. **if** (prepressed) {
46. mPrivateFlags |= PRESSED;
47. refreshDrawableState();
48. postDelayed(mUnsetPressedState,
49. ViewConfiguration.getPressedStateDuration());
50. } **else** **if** (!post(mUnsetPressedState)) {
51. // If the post failed, unpress right now
52. mUnsetPressedState.run();
53. }
54. removeTapCallback();
55. }
56. **break**;
57. **case** MotionEvent.ACTION\_DOWN:
58. **if** (mPendingCheckForTap == **null**) {
59. mPendingCheckForTap = **new** CheckForTap();
60. }
61. mPrivateFlags |= PREPRESSED;
62. mHasPerformedLongPress = **false**;
63. postDelayed(mPendingCheckForTap, ViewConfiguration.getTapTimeout());
64. **break**;
65. **case** MotionEvent.ACTION\_CANCEL:
66. mPrivateFlags &= ~PRESSED;
67. refreshDrawableState();
68. removeTapCallback();
69. **break**;
70. **case** MotionEvent.ACTION\_MOVE:
71. **final** **int** x = (**int**) event.getX();
72. **final** **int** y = (**int**) event.getY();
73. // Be lenient about moving outside of buttons
74. **int** slop = mTouchSlop;
75. **if** ((x < 0 - slop) || (x >= getWidth() + slop) ||
76. (y < 0 - slop) || (y >= getHeight() + slop)) {
77. // Outside button
78. removeTapCallback();
79. **if** ((mPrivateFlags & PRESSED) != 0) {
80. // Remove any future long press/tap checks
81. removeLongPressCallback();
82. // Need to switch from pressed to not pressed
83. mPrivateFlags &= ~PRESSED;
84. refreshDrawableState();
85. }
86. }
87. **break**;
88. }
89. **return** **true**;
90. }
91. **return** **false**;
92. }

相较于刚才的dispatchTouchEvent方法，onTouchEvent方法复杂了很多，不过没关系，我们只挑重点看就可以了。

首先在第14行我们可以看出，如果该控件是可以点击的就会进入到第16行的switch判断中去，而如果当前的事件是抬起手指，则会进入到MotionEvent.ACTION\_UP这个case当中。在经过种种判断之后，会执行到第38行的performClick()方法，那我们进入到这个方法里瞧一瞧：

**[java]** view plain copy

1. **public** **boolean** performClick() {
2. sendAccessibilityEvent(AccessibilityEvent.TYPE\_VIEW\_CLICKED);
3. **if** (mOnClickListener != **null**) {
4. playSoundEffect(SoundEffectConstants.CLICK);
5. mOnClickListener.onClick(**this**);
6. **return** **true**;
7. }
8. **return** **false**;
9. }

可以看到，只要mOnClickListener不是null，就会去调用它的onClick方法，那mOnClickListener又是在哪里赋值的呢？经过寻找后找到如下方法：

**[java]** view plain copy

1. **public** **void** setOnClickListener(OnClickListener l) {
2. **if** (!isClickable()) {
3. setClickable(**true**);
4. }
5. mOnClickListener = l;
6. }

一切都是那么清楚了！当我们通过调用setOnClickListener方法来给控件注册一个点击事件时，就会给mOnClickListener赋值。然后每当控件被点击时，都会在performClick()方法里回调被点击控件的onClick方法。

这样View的整个事件分发的流程就让我们搞清楚了！不过别高兴的太早，现在还没结束，还有一个很重要的知识点需要说明，就是touch事件的层级传递。我们都知道如果给一个控件注册了touch事件，每次点击它的时候都会触发一系列的ACTION\_DOWN，ACTION\_MOVE，ACTION\_UP等事件。这里需要注意，如果你在执行ACTION\_DOWN的时候返回了false，后面一系列其它的action就不会再得到执行了。简单的说，就是当dispatchTouchEvent在进行事件分发的时候，只有前一个action返回true，才会触发后一个action。

说到这里，很多的朋友肯定要有巨大的疑问了。这不是在自相矛盾吗？前面的例子中，明明在onTouch事件里面返回了false，ACTION\_DOWN和ACTION\_UP不是都得到执行了吗？其实你只是被假象所迷惑了，让我们仔细分析一下，在前面的例子当中，我们到底返回的是什么。

参考着我们前面分析的源码，首先在onTouch事件里返回了false，就一定会进入到onTouchEvent方法中，然后我们来看一下onTouchEvent方法的细节。由于我们点击了按钮，就会进入到第14行这个if判断的内部，然后你会发现，不管当前的action是什么，最终都一定会走到第89行，返回一个true。

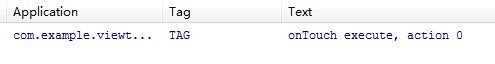
是不是有一种被欺骗的感觉？明明在onTouch事件里返回了false，系统还是在onTouchEvent方法中帮你返回了true。就因为这个原因，才使得前面的例子中ACTION\_UP可以得到执行。

那我们可以换一个控件，将按钮替换成ImageView，然后给它也注册一个touch事件，并返回false。如下所示：

**[java]** view plain copy

1. imageView.setOnTouchListener(**new** OnTouchListener() {
2. @Override
3. **public** **boolean** onTouch(View v, MotionEvent event) {
4. Log.d("TAG", "onTouch execute, action " + event.getAction());
5. **return** **false**;
6. }
7. });

运行一下程序，点击ImageView，你会发现结果如下：



在ACTION\_DOWN执行完后，后面的一系列action都不会得到执行了。这又是为什么呢？因为ImageView和按钮不同，它是默认不可点击的，因此在onTouchEvent的第14行判断时无法进入到if的内部，直接跳到第91行返回了false，也就导致后面其它的action都无法执行了。

好了，关于View的事件分发，我想讲的东西全都在这里了。现在我们再来回顾一下开篇时提到的那三个问题，相信每个人都会有更深一层的理解。

**1. onTouch和onTouchEvent有什么区别，又该如何使用？**

从源码中可以看出，这两个方法都是在View的dispatchTouchEvent中调用的，onTouch优先于onTouchEvent执行。如果在onTouch方法中通过返回true将事件消费掉，onTouchEvent将不会再执行。

另外需要注意的是，onTouch能够得到执行需要两个前提条件，第一mOnTouchListener的值不能为空，第二当前点击的控件必须是enable的。因此如果你有一个控件是非enable的，那么给它注册onTouch事件将永远得不到执行。对于这一类控件，如果我们想要监听它的touch事件，就必须通过在该控件中重写onTouchEvent方法来实现。

**2. 为什么给ListView引入了一个滑动菜单的功能，ListView就不能滚动了？**

如果你阅读了[Android实现图片滚动控件，含页签功能，让你的应用像淘宝一样炫起来](http://blog.csdn.net/sinyu890807/article/details/8744400) 这篇文章。当时我在图片轮播器里使用Button，主要就是因为Button是可点击的，而ImageView是不可点击的。如果想要使用ImageView，可以有两种改法。第一，在ImageView的onTouch方法里返回true，这样可以保证ACTION\_DOWN之后的其它action都能得到执行，才能实现图片滚动的效果。第二，在布局文件里面给ImageView增加一个android:clickable="true"的属性，这样ImageView变成可点击的之后，即使在onTouch里返回了false，ACTION\_DOWN之后的其它action也是可以得到执行的。

### [Android事件分发机制完全解析，带你从源码的角度彻底理解(下)](http://blog.csdn.net/guolin_blog/article/details/9153747)

那么今天我们将继续上次未完成的话题，从源码的角度分析ViewGroup的事件分发。

首先我们来探讨一下，什么是ViewGroup？它和普通的View有什么区别？

顾名思义，ViewGroup就是一组View的集合，它包含很多的子View和子VewGroup，是[**android**](http://lib.csdn.net/base/android)中所有布局的父类或间接父类，像LinearLayout、RelativeLayout等都是继承自ViewGroup的。但ViewGroup实际上也是一个View，只不过比起View，它多了可以包含子View和定义布局参数的功能。ViewGroup继承结构示意图如下所示：



可以看到，我们平时项目里经常用到的各种布局，全都属于ViewGroup的子类。

简单介绍完了ViewGroup，我们现在通过一个Demo来演示一下Android中VewGroup的事件分发流程吧。

首先我们来自定义一个布局，命名为MyLayout，继承自LinearLayout，如下所示：

**[java]** view plain copy

1. **public** **class** MyLayout **extends** LinearLayout {
3. **public** MyLayout(Context context, AttributeSet attrs) {
4. **super**(context, attrs);
5. }
7. }

然后，打开主布局文件activity\_main.xml，在其中加入我们自定义的布局：

**[html]** view plain copy

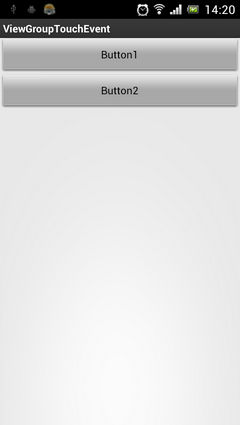
1. **<com.example.viewgrouptouchevent.MyLayout** xmlns:android="http://schemas.android.com/apk/res/android"
2. xmlns:tools="http://schemas.android.com/tools"
3. android:id="@+id/my\_layout"
4. android:layout\_width="match\_parent"
5. android:layout\_height="match\_parent"
6. android:orientation="vertical" **>**
8. **<Button**
9. android:id="@+id/button1"
10. android:layout\_width="match\_parent"
11. android:layout\_height="wrap\_content"
12. android:text="Button1" **/>**
14. **<Button**
15. android:id="@+id/button2"
16. android:layout\_width="match\_parent"
17. android:layout\_height="wrap\_content"
18. android:text="Button2" **/>**
20. **</com.example.viewgrouptouchevent.MyLayout>**

可以看到，我们在MyLayout中添加了两个按钮，接着在MainActivity中为这两个按钮和MyLayout都注册了监听事件：

**[java]** view plain copy

1. myLayout.setOnTouchListener(**new** OnTouchListener() {
2. @Override
3. **public** **boolean** onTouch(View v, MotionEvent event) {
4. Log.d("TAG", "myLayout on touch");
5. **return** **false**;
6. }
7. });
8. button1.setOnClickListener(**new** OnClickListener() {
9. @Override
10. **public** **void** onClick(View v) {
11. Log.d("TAG", "You clicked button1");
12. }
13. });
14. button2.setOnClickListener(**new** OnClickListener() {
15. @Override
16. **public** **void** onClick(View v) {
17. Log.d("TAG", "You clicked button2");
18. }
19. });

我们在MyLayout的onTouch方法，和Button1、Button2的onClick方法中都打印了一句话。现在运行一下项目，效果图如下所示：



分别点击一下Button1、Button2和空白区域，打印结果如下所示：



你会发现，当点击按钮的时候，MyLayout注册的onTouch方法并不会执行，只有点击空白区域的时候才会执行该方法。你可以先理解成Button的onClick方法将事件消费掉了，因此事件不会再继续向下传递。

那就说明Android中的touch事件是先传递到View，再传递到ViewGroup的？现在下结论还未免过早了，让我们再来做一个实验。

查阅文档可以看到，ViewGroup中有一个onInterceptTouchEvent方法，我们来看一下这个方法的源码：

**[java]** view plain copy

1. /\*\*
2. \* Implement this method to intercept all touch screen motion events.  This
3. \* allows you to watch events as they are dispatched to your children, and
4. \* take ownership of the current gesture at any point.
5. \*
6. \* <p>Using this function takes some care, as it has a fairly complicated
7. \* interaction with {@link View#onTouchEvent(MotionEvent)
8. \* View.onTouchEvent(MotionEvent)}, and using it requires implementing
9. \* that method as well as this one in the correct way.  Events will be
10. \* received in the following order:
11. \*
12. \* <ol>
13. \* <li> You will receive the down event here.
14. \* <li> The down event will be handled either by a child of this view
15. \* group, or given to your own onTouchEvent() method to handle; this means
16. \* you should implement onTouchEvent() to return true, so you will
17. \* continue to see the rest of the gesture (instead of looking for
18. \* a parent view to handle it).  Also, by returning true from
19. \* onTouchEvent(), you will not receive any following
20. \* events in onInterceptTouchEvent() and all touch processing must
21. \* happen in onTouchEvent() like normal.
22. \* <li> For as long as you return false from this function, each following
23. \* event (up to and including the final up) will be delivered first here
24. \* and then to the target's onTouchEvent().
25. \* <li> If you return true from here, you will not receive any
26. \* following events: the target view will receive the same event but
27. \* with the action {@link MotionEvent#ACTION\_CANCEL}, and all further
28. \* events will be delivered to your onTouchEvent() method and no longer
29. \* appear here.
30. \* </ol>
31. \*
32. \* @param ev The motion event being dispatched down the hierarchy.
33. \* @return Return true to steal motion events from the children and have
34. \* them dispatched to this ViewGroup through onTouchEvent().
35. \* The current target will receive an ACTION\_CANCEL event, and no further
36. \* messages will be delivered here.
37. \*/
38. **public** **boolean** onInterceptTouchEvent(MotionEvent ev) {
39. **return** **false**;
40. }

如果不看源码你还真可能被这注释吓到了，这么长的英文注释看得头都大了。可是源码竟然如此简单！只有一行代码，返回了一个false！

好吧，既然是布尔型的返回，那么只有两种可能，我们在MyLayout中重写这个方法，然后返回一个true试试，代码如下所示：

**[java]** view plain copy

1. **public** **class** MyLayout **extends** LinearLayout {
3. **public** MyLayout(Context context, AttributeSet attrs) {
4. **super**(context, attrs);
5. }
7. @Override
8. **public** **boolean** onInterceptTouchEvent(MotionEvent ev) {
9. **return** **true**;
10. }
12. }

现在再次运行项目，然后分别Button1、Button2和空白区域，打印结果如下所示：



你会发现，不管你点击哪里，永远都只会触发MyLayout的touch事件了，按钮的点击事件完全被屏蔽掉了！这是为什么呢？如果Android中的touch事件是先传递到View，再传递到ViewGroup的，那么MyLayout又怎么可能屏蔽掉Button的点击事件呢？

看来只有通过阅读源码，搞清楚Android中ViewGroup的事件分发机制，才能解决我们心中的疑惑了，不过这里我想先跟你透露一句，Android中touch事件的传递，绝对是先传递到ViewGroup，再传递到View的。记得在[**Android事件分发机制完全解析，带你从源码的角度彻底理解(上)**](http://blog.csdn.net/sinyu890807/article/details/9097463) 中我有说明过，只要你触摸了任何控件，就一定会调用该控件的dispatchTouchEvent方法。这个说法没错，只不过还不完整而已。实际情况是，当你点击了某个控件，首先会去调用该控件所在布局的dispatchTouchEvent方法，然后在布局的dispatchTouchEvent方法中找到被点击的相应控件，再去调用该控件的dispatchTouchEvent方法。如果我们点击了MyLayout中的按钮，会先去调用MyLayout的dispatchTouchEvent方法，可是你会发现MyLayout中并没有这个方法。那就再到它的父类LinearLayout中找一找，发现也没有这个方法。那只好继续再找LinearLayout的父类ViewGroup，你终于在ViewGroup中看到了这个方法，按钮的dispatchTouchEvent方法就是在这里调用的。修改后的示意图如下所示：



那还等什么？快去看一看ViewGroup中的dispatchTouchEvent方法的源码吧！代码如下所示：

**[java]** view plain copy

1. **public** **boolean** dispatchTouchEvent(MotionEvent ev) {
2. **final** **int** action = ev.getAction();
3. **final** **float** xf = ev.getX();
4. **final** **float** yf = ev.getY();
5. **final** **float** scrolledXFloat = xf + mScrollX;
6. **final** **float** scrolledYFloat = yf + mScrollY;
7. **final** Rect frame = mTempRect;
8. **boolean** disallowIntercept = (mGroupFlags & FLAG\_DISALLOW\_INTERCEPT) != 0;
9. **if** (action == MotionEvent.ACTION\_DOWN) {
10. **if** (mMotionTarget != **null**) {
11. mMotionTarget = **null**;
12. }
13. **if** (disallowIntercept || !onInterceptTouchEvent(ev)) {
14. ev.setAction(MotionEvent.ACTION\_DOWN);
15. **final** **int** scrolledXInt = (**int**) scrolledXFloat;
16. **final** **int** scrolledYInt = (**int**) scrolledYFloat;
17. **final** View[] children = mChildren;
18. **final** **int** count = mChildrenCount;
19. **for** (**int** i = count - 1; i >= 0; i--) {
20. **final** View child = children[i];
21. **if** ((child.mViewFlags & VISIBILITY\_MASK) == VISIBLE
22. || child.getAnimation() != **null**) {
23. child.getHitRect(frame);
24. **if** (frame.contains(scrolledXInt, scrolledYInt)) {
25. **final** **float** xc = scrolledXFloat - child.mLeft;
26. **final** **float** yc = scrolledYFloat - child.mTop;
27. ev.setLocation(xc, yc);
28. child.mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;
29. **if** (child.dispatchTouchEvent(ev))  {
30. mMotionTarget = child;
31. **return** **true**;
32. }
33. }
34. }
35. }
36. }
37. }
38. **boolean** isUpOrCancel = (action == MotionEvent.ACTION\_UP) ||
39. (action == MotionEvent.ACTION\_CANCEL);
40. **if** (isUpOrCancel) {
41. mGroupFlags &= ~FLAG\_DISALLOW\_INTERCEPT;
42. }
43. **final** View target = mMotionTarget;
44. **if** (target == **null**) {
45. ev.setLocation(xf, yf);
46. **if** ((mPrivateFlags & CANCEL\_NEXT\_UP\_EVENT) != 0) {
47. ev.setAction(MotionEvent.ACTION\_CANCEL);
48. mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;
49. }
50. **return** **super**.dispatchTouchEvent(ev);
51. }
52. **if** (!disallowIntercept && onInterceptTouchEvent(ev)) {
53. **final** **float** xc = scrolledXFloat - (**float**) target.mLeft;
54. **final** **float** yc = scrolledYFloat - (**float**) target.mTop;
55. mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;
56. ev.setAction(MotionEvent.ACTION\_CANCEL);
57. ev.setLocation(xc, yc);
58. **if** (!target.dispatchTouchEvent(ev)) {
59. }
60. mMotionTarget = **null**;
61. **return** **true**;
62. }
63. **if** (isUpOrCancel) {
64. mMotionTarget = **null**;
65. }
66. **final** **float** xc = scrolledXFloat - (**float**) target.mLeft;
67. **final** **float** yc = scrolledYFloat - (**float**) target.mTop;
68. ev.setLocation(xc, yc);
69. **if** ((target.mPrivateFlags & CANCEL\_NEXT\_UP\_EVENT) != 0) {
70. ev.setAction(MotionEvent.ACTION\_CANCEL);
71. target.mPrivateFlags &= ~CANCEL\_NEXT\_UP\_EVENT;
72. mMotionTarget = **null**;
73. }
74. **return** target.dispatchTouchEvent(ev);
75. }

这个方法代码比较长，我们只挑重点看。首先在第13行可以看到一个条件判断，如果disallowIntercept和!onInterceptTouchEvent(ev)两者有一个为true，就会进入到这个条件判断中。disallowIntercept是指是否禁用掉事件拦截的功能，默认是false，也可以通过调用requestDisallowInterceptTouchEvent方法对这个值进行修改。那么当第一个值为false的时候就会完全依赖第二个值来决定是否可以进入到条件判断的内部，第二个值是什么呢？竟然就是对onInterceptTouchEvent方法的返回值取反！也就是说如果我们在onInterceptTouchEvent方法中返回false，就会让第二个值为true，从而进入到条件判断的内部，如果我们在onInterceptTouchEvent方法中返回true，就会让第二个值为false，从而跳出了这个条件判断。

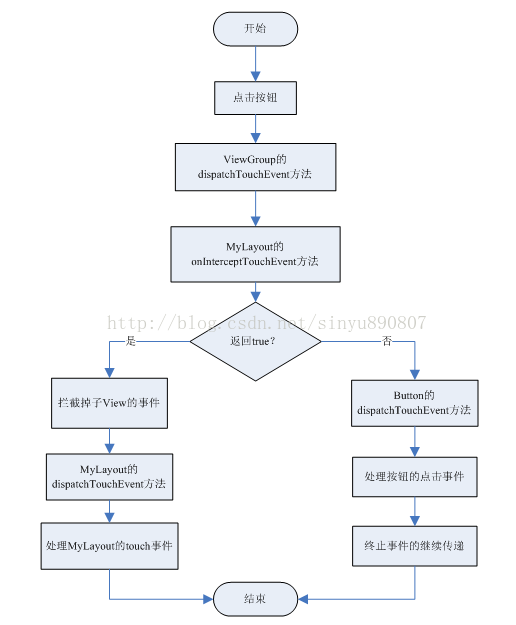
这个时候你就可以思考一下了，由于我们刚刚在MyLayout中重写了onInterceptTouchEvent方法，让这个方法返回true，导致所有按钮的点击事件都被屏蔽了，那我们就完全有理由相信，按钮点击事件的处理就是在第13行条件判断的内部进行的！

那我们重点来看下条件判断的内部是怎么实现的。在第19行通过一个for循环，遍历了当前ViewGroup下的所有子View，然后在第24行判断当前遍历的View是不是正在点击的View，如果是的话就会进入到该条件判断的内部，然后在第29行调用了该View的dispatchTouchEvent，之后的流程就和 [Android事件分发机制完全解析，带你从源码的角度彻底理解(上)](http://blog.csdn.net/sinyu890807/article/details/9097463) 中讲解的是一样的了。我们也因此证实了，按钮点击事件的处理确实就是在这里进行的。

然后需要注意一下，调用子View的dispatchTouchEvent后是有返回值的。我们已经知道，如果一个控件是可点击的，那么点击该控件时，dispatchTouchEvent的返回值必定是true。因此会导致第29行的条件判断成立，于是在第31行给ViewGroup的dispatchTouchEvent方法直接返回了true。这样就导致后面的代码无法执行到了，也是印证了我们前面的Demo打印的结果，如果按钮的点击事件得到执行，就会把MyLayout的touch事件拦截掉。

那如果我们点击的不是按钮，而是空白区域呢？这种情况就一定不会在第31行返回true了，而是会继续执行后面的代码。那我们继续往后看，在第44行，如果target等于null，就会进入到该条件判断内部，这里一般情况下target都会是null，因此会在第50行调用super.dispatchTouchEvent(ev)。这句代码会调用到哪里呢？当然是View中的dispatchTouchEvent方法了，因为ViewGroup的父类就是View。之后的处理逻辑又和前面所说的是一样的了，也因此MyLayout中注册的onTouch方法会得到执行。之后的代码在一般情况下是走不到的了，我们也就不再继续往下分析。

再看一下整个ViewGroup事件分发过程的流程图吧，相信可以帮助大家更好地去理解：



现在整个ViewGroup的事件分发流程的分析也就到此结束了，我们最后再来简单梳理一下吧。

1. Android事件分发是先传递到ViewGroup，再由ViewGroup传递到View的。

2. 在ViewGroup中可以通过onInterceptTouchEvent方法对事件传递进行拦截，onInterceptTouchEvent方法返回true代表不允许事件继续向子View传递，返回false代表不对事件进行拦截，默认返回false。

3. 子View中如果将传递的事件消费掉，ViewGroup中将无法接收到任何事件。

好了，Android事件分发机制完全解析到此全部结束

### 3.静态和单例之间的区别和针对项目的选择

第一种回答

两者占内存不一样  
没记错的话，类加载的时候都要加载类的定义；执行方法的时候都要占用栈；但是单例需要在堆里面new一个内存空间出来，静态方法ms不用。

我的习惯用法是不需要依赖于其它类或资源时，用静态方法，这时就是一个面向过程的函数而已；如果需要依赖其它类的实例，或者需要某些资源时，用单例。

=======2015-8-13========  
补充一下。  
如果你的逻辑有需要重载的可能性，那么用单例。  
如果肯定不会重载，那么用静态方法。

有个更简单的判断方法，如果是业务相关逻辑，绝对不用静态方法。如果是业务无关的（绝对无关的），那么可以用静态。

### 4.HashMap和HashTable

**区别：**  
1.HashTable是Dictionary的子类,HashMap是Map接口的一个实现类;

2.HashTable中的方法是同步的,而HashMap中方法是非同步的.也就是说,在多线程的情况下用HashMap需要额外的同步机制.

Map Collections.synchronziedMap(Map m)这个方法返回一个同步的Map,封装了底层的HashMap方法,使得多线程安全.

或者采用ConcurrentMap接口;

3.HashMap中,键和值都可以为null(null键只能有一个)，HashTable不允许为null。当get()方法时返回null，即表示没有该键，也可以表示该键对应的值为null。因此判断HashMap里是否存在某个键时，不能用get()方法，应该用containsKey()方法

**相同：**

1.有两个参数影响性能：初始容量和加载因子。

初始容量：哈希表创建时的容量，初始容量设置太高可能会浪费空间；

加载因子：对哈希表在其容量自动增加之前可以达到多满的一个尺度（默认为0.75），加载因子过高虽然减少了空间开销，但同时也增加了查询成本。

当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 *rehash* 操作（即重建内部数据结构）。

2.所有类的“Collection视图方法”返回的Collection的iterator方法返回的迭代器是快速失败的。

**二、HashMap中key和value的原理**

HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。HashMap底层就是一个数组结构，数组中的每一项又是一个链表。当新建一个HashMap的时候，就会初始化一个数组。每个Map.Entry是一个Key-Value对，也是数组中的元素，它持有指向下一个元素的引用，这就构成了链表。  HashMap的存取实现：

   1) 存储：

　　当我们往HashMap中put元素的时候，先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置（即下标），如果数组该位置上已经存放有其他元素了，那么在这个位置上的元素将以链表的形式存放，新加入的放在链头，最先加入的放在链尾。如果数组该位置上没有元素，就直接将该元素放到此数组中的该位置上。

当系统决定存储HashMap中的key-value对时，完全没有考虑Entry中的value，仅仅只是根据key来计算并决定每个Entry的存储位置。我们完全可以把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

   hash(int h)--计算hash值的方法根据key的hashCode重新计算一次散列。此算法加入了高位计算，防止低位不变，高位变化时，造成的hash冲突。

  2) 读取：

　　从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

3）Fail-Fast机制：

   我们知道java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

   这一策略在源码中的实现是通过modCount域，modCount顾名思义就是修改次数，对HashMap内容的修改都将增加这个值，那么在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。在迭代过程中，判断modCount跟expectedModCount是否相等，如果不相等就表示已经有其他线程修改了Map： 注意到modCount声明为volatile，保证线程之间修改的可见性。

2、hash算法   
我们可以看到在hashmap中要找到某个元素，需要根据key的hash值来求得对应数组中的位置。如何计算这个位置就是hash算法。前面说过hashmap的数据结构是数组和链表的结合，所以我们当然希望这个hashmap里面的元素位置尽量的分布均匀些，尽量使得每个位置上的元素数量只有一个，那么当我们用hash算法求得这个位置的时候，马上就可以知道对应位置的元素就是我们要的，而不用再去遍历链表。

### 5.Mvc mvp的区别

> MVC/MVP

  View强依赖于Model是MVC的主要问题。由此导致很多控件都是根据业务定制，从Android的角度来看，原本可以由一个通用的layout就能实现的控件，由于要绑定实体模型，现在必须要自定义控件，这导致出现大量不必要的重复代码。因此有必要将View和Model进行解耦，而MVP的主要思想就是解耦View和Model。由此引入MVP就显得很自然。

  MVP存在的问题:

尽管已经有了大量的应用，但不可否认该模式的还是存在一些问题，这些问题在携程的使用过程中也得到了体现。比如，上下文丢失问题，生命周期问题，内存泄露问题以及大量的自定义接口，回调链变长等问题。可以归纳为：

1.业务复杂时，可能使得Activity变成更加复杂，比如要实现N个IView，然后写更多个模版方法。

2.业务复杂时，各个角色之间通信会变得很冗长和复杂，回调链过长。

3.Presenter处理业务，让业务变得很分散，不能全局掌握业务，很难去回答某个业务究竟是在哪里处理的。

4.用Presenter替代Controller是一个危险的做法，可能出现内存泄漏，生命周期不同步，上下文丢失等问题。

与mvp相关的开源项目：https://github.com/yaozs/YzsBaseActivity  https://github.com/yaozs/YzsLib

框架模式MVC与MVP在Android中的应用: <http://blog.csdn.net/gjnm820/article/details/51733361>

Google推出的基于MVP架构的demo,基于此架构我在GitHub上开源了一个项目MinimalistWeather

Android MVP框架MVPro的使用和源码分析- http://blog.csdn.net/qibin0506/article/details/49992897

浅谈Andorid开发中的MVP模式- http://blog.csdn.net/loongggdroid/article/details/50592777

1.MVC的基本介绍

MVC全称是Model - View - Controller，是模型(model)－视图(view)－控制器(controller)的缩写。MVC是一种框架模式而非设计模式，GOF把MVC看作是3种设计模式：观察者模式、策略模式与组合模式的合体，而核心是观察者模式。简而言之，框架是大智慧，用来对软件设计进行分工；设计模式是小技巧，对具体问题提出解决方案，以提高代码复用率，降低耦合度。

使用 MVC，把业务逻辑抽离到 Controller 中，让 View 层专注于显示 UI。

1.MVC的优点

（1）首先就是理解比较容易，技术含量不高，这对开发和维护来说成本较低也易于维护与修改。

（2）耦合性不高，表现层与业务层分离各司其职，对开发来说很有利。

2.MVC的缺点

（1）完全理解MVC并不是很容易。使用MVC需要精心的计划，由于它的内部原理比较复杂，所以需要花费一些时间去思考。同时由于模型和视图要严格的分离，这样也给调试应用程序带来了一定的困难。每个构件在使用之前都需要经过彻底的[**测试**](http://lib.csdn.net/base/softwaretest)。

（2）对于小项目，MVC反而会带来更大的工作量以及复杂性。

2.MVC在Android中的应用

[**Android**](http://lib.csdn.net/base/android)中对MVC的应用很经典，在[**android**](http://lib.csdn.net/base/android)中视图View层一般采用XML文件进行界面的描述。如下例子：

<?xml version="1.0" encoding="utf-8"?>

<TextView xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="50dp"

android:id="@+id/list\_item\_text"

android:textSize="16sp"

android:gravity="left|center\_vertical"

android:padding="10dp" />

而对于模型Model部分则大多对应于本地的数据文件或网络获取的数据体，很多情况下我们对这些数据的处理也会在这一层中进行。

最后的控制器Controller则当之无愧的是右Activity承担。

3.总结

虽说上面的介绍中我们感受到Android在MVC方面的结构，但是，这个框架并非我们自己完成的，而是由framework给我们搭建好的并提供给我们，在平时的开发中，特别是用Android开发，我们并不常用到MVC模式去脱离Android UI系统构建自己的框架结构。

------------------------------------

1.MVP介绍

MVP模式是MVC模式的一个演化版本，MVP全称Model-View-Presenter。目前MVP在Android应用开发中越来越重要了。

在Android中，业务逻辑和数据存取是紧紧耦合的，很多缺乏经验的开发者很可能会将各种各样的业务逻辑塞进某个Activity、Fragment或者自定义View中，这样会使得这些组件的单个类型臃肿不堪。如果不将具体的业务逻辑抽离出来，当UI变化时，你就需要去原来的View中抽离具体业务逻辑，这必然会很麻烦并且易出错。

2.使用MVP的好处

（1）MVP模式会解除View与Model的耦合，有效的降低View的复杂性。同时又带来了良好的可扩展性、可测试性，保证系统的整洁性和灵活性。

（2）MVP模式可以分离显示层与逻辑层，它们之间通过接口进行通信，降低耦合。理想化的MVP模式可以实现同一份逻辑代码搭配不同的显示界面，因为它们之间并不依赖与具体，而是依赖于抽象。这使得Presenter可以运用于任何实现了View逻辑接口的UI，使之具有更广泛的适用性，保证了灵活度。

3.MVP模式的三个角色

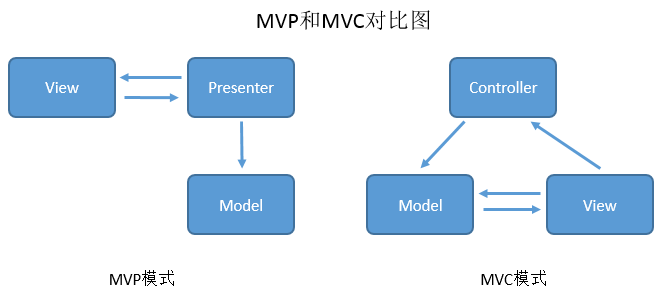
（1）Presenter – 交互中间人：Presenter主要作为沟通View与Model的桥梁，它从Model层检索数据后，返回给View层，使得View与Model之间没有耦合，也将业务逻辑从View角色上抽离出来。

（2）View – 用户界面：View通常是指Activity、Fragment或者某个View控件，它含有一个Presenter成员变量。通常View需要实现一个逻辑接口，将View上的操作转交给Presenter进行实现，最后，Presenter 调用View逻辑接口将结果返回给View元素。

（3）Model – 数据的存取：Model 角色主要是提供数据的存取功能。Presenter 需要通过Model层存储、获取数据，Model就像一个数据仓库。更直白的说，Model是封装了数据库DAO或者网络获取数据的角色，或者两种数据方式获取的集合。

4.与MVC、MVVM的区别

1.与MVC的区别



从上图可以看出：MVC的耦合性还是较高的，View可以直接访问Model，导致3者之间构成了回路。所以两者的主要区别是，MVP中View不能直接访问Model，需要通过Presenter发出请求，View与Model不能直接通信。

2.与MVVM（Model-View-ViewModel）的区别

MVVM与MVP非常相似，唯一区别是View和Model进行双向绑定，两者之间有一方发生变化则会反应到另一方上。MVVM模式有点像ListView与Adapter、数据集的关系，当数据集发生变化时，调用Adapter的notifyDataSetChanged之后View就直接更新，同时它们之间又没有耦合，使得ListView变得更加灵活。

5.MVP简单实现

可以参考：

1.[androidmvp](https://github.com/antoniolg/androidmvp)

2.[archi](https://github.com/ivacf/archi)

6.MVP 与Activity、Fragment的生命周期

由于Presenter 经常性的持有Activity 的强引用，如果在一些请求结束之前Activity 被销毁了，那么Presenter 一直持有Activity 对象，使得Activity 对象无法回收，此时就会发生内存泄露。那么解决方法就是采用弱引用和Activity、Fragment的生命周期来解决这个问题。首先建立一个Presenter 抽象：

public abstract class BasePresenter<T> {

protected Reference<T> mViewRef; //View接口类型的弱引用

public void attachView(T view){

mViewRef = new WeakReference<T>(view); //建立关联

}

protected T getView(){

return mViewRef.get(); //获取View

}

public boolean isViewAttached(){

return mViewRef != null && mViewRef.get() != null; //判断是否与View建立关联

}

public void detachView(){

if(mViewRef != null){

mViewRef.clear(); //解除关联

mViewRef = null;

}

}

}

通常这个View类型应该就是实现了某个特定接口的Activity或者Fragment等类型。

创建一个MVPBaseActivity基类，通过这个基类声明周期函数来控制它与Presenter 的关系。代码如下：

public abstract class MVPBaseActivity<V, T extends BasePresenter<V>> extends Activity {

protected T mPresenter; //Presenter对象

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mPresenter = createPresenter();

mPresenter.attachView((V)this);

}

@Override

protected void onDestroy() {

super.onDestroy();

mPresenter.detachView();

}

protected abstract T createPresenter();

}

MVPBaseActivity含有两个泛型，一个是View的接口类型，一个是Presenter的具体类型。

### 6.picasso源码解析 及其 核心

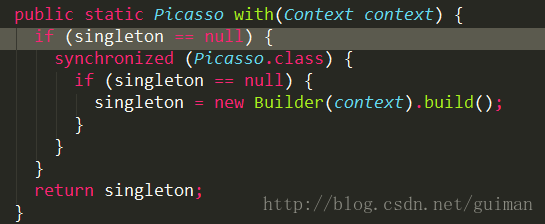
下面我们从源码的角度来解析下它到底是怎么实现我们的图片加载和使用的。

以下面最简洁的加载为示例：

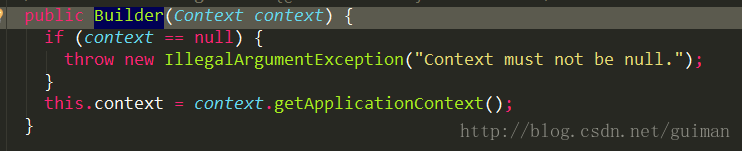
Picasso.with(MainActivity.this).load(url).into(headerImage);

with

首先Picasso会调用静态with方法，那么我们来看看with方法是怎么实现的：



由上面的源码我们可以看到，在with方法中主要做了一件事，那就是返回一个Picasso实例，当然这个实例也不是那么简单的创建的，为了防止Picasso的多次创建，这里使用了双重加锁的单例模式来创建的，主要目的是为了保证线程的安全性。但是它又不是直接的使用单例模式创建的，在创建实例的过程中使用了Builder模式，它可以使Picasso在创建时初始化很多对象，以便后期使用，那么我们就来看看这个Builder是怎么操作的：



在Builder的构造方法中就只是获取到当前应用级别的上下文，也就说明了Picasso是针对应用级别的使用，不会是随着Activity或是Fragment的生命周期而产生变化，只有当当前的应用退出或是销毁时Picasso才会停止它的行为。

那么接下来看下build方法中做了哪些操作呢：



这里代码也是很简单，主要是初始化了downloader，cache，service，dispatcher等几个实例变量，而这几个变量值也是已设置的，如源码：

public Builder downloader(Downloader downloader) {

if (downloader == null) {

throw new IllegalArgumentException("Downloader must not be null.");

}

if (this.downloader != null) {

throw new IllegalStateException("Downloader already set.");

}

this.downloader = downloader;

return this;

}

public Builder executor(ExecutorService executorService) {

if (executorService == null) {

throw new IllegalArgumentException("Executor service must not be null.");

}

if (this.service != null) {

throw new IllegalStateException("Executor service already set.");

}

this.service = executorService;

return this;

}

public Builder memoryCache(Cache memoryCache) {

if (memoryCache == null) {

throw new IllegalArgumentException("Memory cache must not be null.");

}

if (this.cache != null) {

throw new IllegalStateException("Memory cache already set.");

}

this.cache = memoryCache;

return this;

}

这些设置就像我们平常使用AlertDialog一样，货到到Builder之后分别进行设置就行。

ok，我们现在来看看初始化中几个非常重要的变量：

downloader 下载器

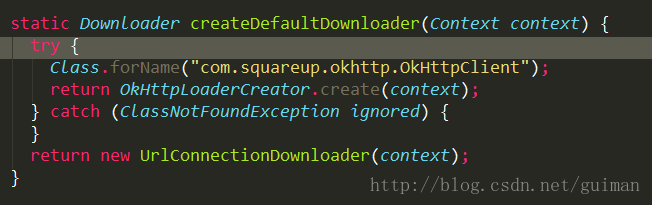
首先看下downloader，builder中首先判断downloader是否为空值，当为空值时就为它初始化默认值，如：

if (downloader == null) {

downloader = Utils.createDefaultDownloader(context);

}

来看下Utils中是怎么实现downloader 的初始化的：



createDefaultDownloader方法中首先使用[Java](http://lib.csdn.net/base/java)反射机制来查找项目中是否使用了okhttp网络加载框架，如果使用了则会使用okhttp作为图片的加载方式，如果没有使用，则会使用内置的封装加载器UrlConnectionDownloader。

注：由于okhttp3的包名已更换，所以在这里都是使用内置的封装下载器，这个是一个小bug等待完善。当修复之后Picasso+okhttp3则是最理想的加载方式。

service 线程池

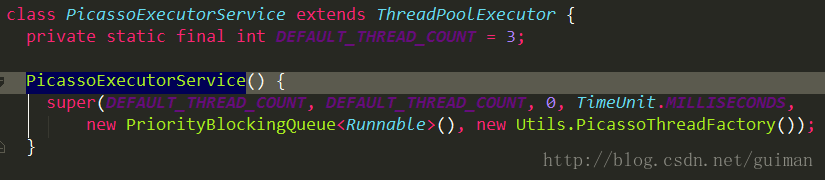
同样的使用，首先判断是否为空，如果为空则初始化默认对象：

if (service == null) {

service = new PicassoExecutorService();

}

我们在来看看PicassoExecutorService的源码：



PicassoExecutorService直接继承与ThreadPoolExecutor线程池，在构造方法中初始化了主线程大小，最大线程等，如：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory) {

if (corePoolSize < 0 ||

maximumPoolSize <= 0 ||

maximumPoolSize < corePoolSize ||

keepAliveTime < 0)

throw new IllegalArgumentException();

if (workQueue == null || threadFactory == null || handler == null)

throw new NullPointerException();

this.corePoolSize = corePoolSize;

this.maximumPoolSize = maximumPoolSize;

this.workQueue = workQueue;

this.keepAliveTime = unit.toNanos(keepAliveTime);

this.threadFactory = threadFactory;

this.handler = handler;

当然，在Picasso线程池中主线程和最大线程数是可变的，根据用户使用的网络类型来设置线程数量，后面会详细说明它的应用。

dispatcher 事务分发器

dispatcher 在Picasso中扮演着十分重要的角色，可以说它是整个图片加载过程中的中转站，主线程和子线程来回的切换主要都是依赖它的存在。下面我们将来仔细的研究下它的设计，理解好它的存在对整个Picasso框架的理解也基本明朗。

首先，来看看它的登场亮相：

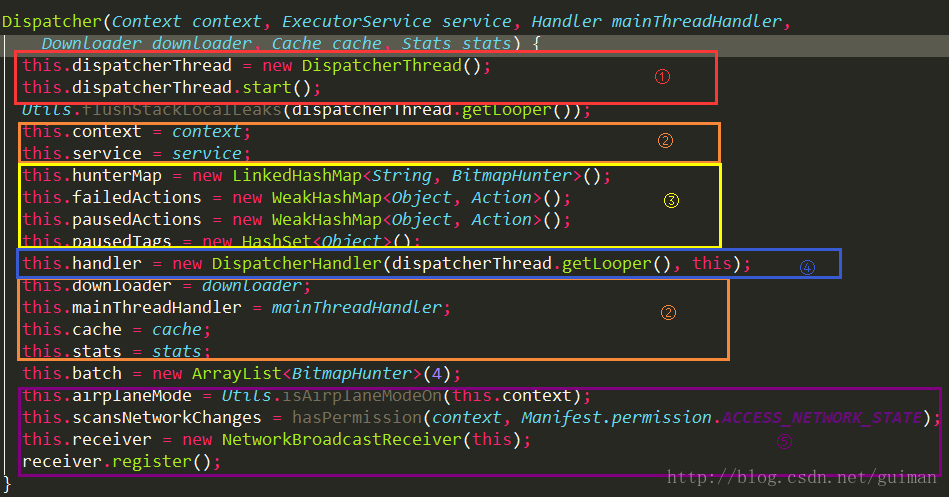
Dispatcher dispatcher = new Dispatcher(context, service, HANDLER, downloader, cache, stats);

1. 1
2. 1

它的登场就是创建一个Dispatcher 对象实例，当然，它传递了在Builder中所初始化的对象实例，比如downloader下载器，用于图片的下载，当然下载是不能再主线程进行的，所以这里也传递了service线程池，而下载好的资源也是不能直接在子线程中进行更新UI的，所以同时也把主线程中的HANDLER传递进去，同时应用级别的上下文context，和缓存cache，状态变化stats等也传递进去进行相对应的业务操作。

经过上面的分析，我们可以很清楚的看出，就这么简单的一个创建实例已经很明确的表达出了Dispatcher存在的意义，而且我们也明确了它大概的职责。

那么我们接着看看Dispatcher的构造方法中具体的做了哪些操作：



在Dispatcher的构造方法中我把它分为了5个部分，下面来详细的解析下：

①：dispatcherThread，它是一个HandlerThread线程，如：

static class DispatcherThread extends HandlerThread {

DispatcherThread() {

super(Utils.THREAD\_PREFIX + DISPATCHER\_THREAD\_NAME, THREAD\_PRIORITY\_BACKGROUND);

}

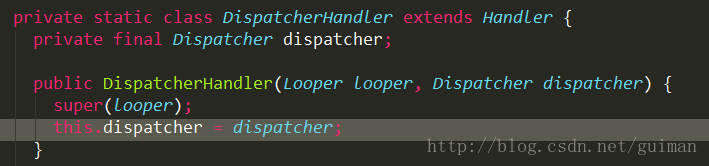
}

它的创建主要是开启一个子线程供Dispatcher调用，目的就是为了在子线程中去执行耗时的图片下载操作。

②：对象实例的接受，主要是接受在Picasso中初始化的对象实例，这个没有什么好说的。

③：创建用于保存数据、对象的集合，也就是为了保存对象或状态用的。

④：DispatcherHandler，这是一个Handler，并且是作用在dispatcherThread线程中的Handler，它用于把在dispatcherThread子线程的操作转到到Dispatcher中去，它的构造方法中接受了Dispatcher对象：



我们在来看看他的handleMessage方法是怎么处理消息的：

@Override public void handleMessage(final Message msg) {

switch (msg.what) {

case REQUEST\_SUBMIT: {

Action action = (Action) msg.obj;

dispatcher.performSubmit(action);

break;

}

case REQUEST\_CANCEL: {

Action action = (Action) msg.obj;

dispatcher.performCancel(action);

break;

}

case TAG\_PAUSE: {

Object tag = msg.obj;

dispatcher.performPauseTag(tag);

break;

}

case TAG\_RESUME: {

Object tag = msg.obj;

dispatcher.performResumeTag(tag);

break;

}

case HUNTER\_COMPLETE: {

BitmapHunter hunter = (BitmapHunter) msg.obj;

dispatcher.performComplete(hunter);

break;

}

case HUNTER\_RETRY: {

BitmapHunter hunter = (BitmapHunter) msg.obj;

dispatcher.performRetry(hunter);

break;

}

case HUNTER\_DECODE\_FAILED: {

BitmapHunter hunter = (BitmapHunter) msg.obj;

dispatcher.performError(hunter, false);

break;

}

case HUNTER\_DELAY\_NEXT\_BATCH: {

dispatcher.performBatchComplete();

break;

}

case NETWORK\_STATE\_CHANGE: {

NetworkInfo info = (NetworkInfo) msg.obj;

dispatcher.performNetworkStateChange(info);

break;

}

case AIRPLANE\_MODE\_CHANGE: {

dispatcher.performAirplaneModeChange(msg.arg1 == AIRPLANE\_MODE\_ON);

break;

}

default:

Picasso.HANDLER.post(new Runnable() {

@Override public void run() {

throw new AssertionError("Unknown handler message received: " + msg.what);

}

});

}

}

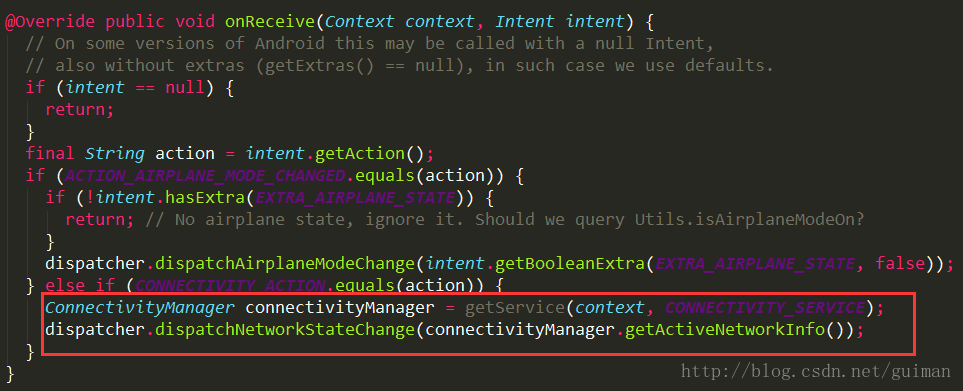
而从它的处理消息的handleMessage中我们可以看出，所有的消息并没有被直接进行处理而是转移到了dispatcher中，在dispatcher中进行相应的处理。

⑤：监听网络变化操作，它是用于监听用户手机网络变化而存在的。我们主要来看看NetworkBroadcastReceiver这个类：



在构造参数中也接收到Dispatcher对象，并有注册广播和销毁广播的方法，当然它也没有直接的处理，也是传递到Dispatcher中进行消化的。

我们在来看看当用户的网络发生变化时，它会做哪些操作：



我们可以看到主要分为两个，一个是航班模式，一个是正常的网络状态变化，航班模式我们先不用理会，主要看下网络变化的操作：

void dispatchNetworkStateChange(NetworkInfo info) {

handler.sendMessage(handler.obtainMessage(NETWORK\_STATE\_CHANGE, info));

}

这里的handler就是DispatcherHandler，那么我们看看它又是怎么做的：

case NETWORK\_STATE\_CHANGE: {

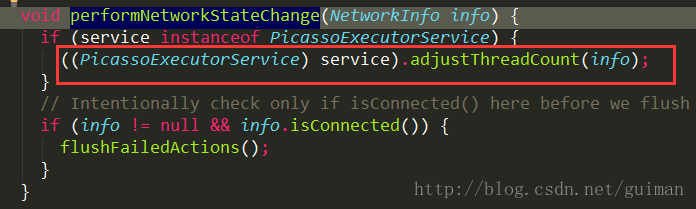
NetworkInfo info = (NetworkInfo) msg.obj;

dispatcher.performNetworkStateChange(info);

break;

}

调用dispatcher的performNetworkStateChange方法来处理：



当网络变化时，它会传递给我们的PicassoExecutorService线程池，在adjustThreadCount方法中判断用户是使用的那类型网络，如wifi，4G等，然后为线程池设置相应的线程数。来看：

void adjustThreadCount(NetworkInfo info) {

if (info == null || !info.isConnectedOrConnecting()) {

setThreadCount(DEFAULT\_THREAD\_COUNT);

return;

}

switch (info.getType()) {

case ConnectivityManager.TYPE\_WIFI:

case ConnectivityManager.TYPE\_WIMAX:

case ConnectivityManager.TYPE\_ETHERNET:

setThreadCount(4);

break;

case ConnectivityManager.TYPE\_MOBILE:

switch (info.getSubtype()) {

case TelephonyManager.NETWORK\_TYPE\_LTE: // 4G

case TelephonyManager.NETWORK\_TYPE\_HSPAP:

case TelephonyManager.NETWORK\_TYPE\_EHRPD:

setThreadCount(3);

break;

case TelephonyManager.NETWORK\_TYPE\_UMTS: // 3G

case TelephonyManager.NETWORK\_TYPE\_CDMA:

case TelephonyManager.NETWORK\_TYPE\_EVDO\_0:

case TelephonyManager.NETWORK\_TYPE\_EVDO\_A:

case TelephonyManager.NETWORK\_TYPE\_EVDO\_B:

setThreadCount(2);

break;

case TelephonyManager.NETWORK\_TYPE\_GPRS: // 2G

case TelephonyManager.NETWORK\_TYPE\_EDGE:

setThreadCount(1);

break;

default:

setThreadCount(DEFAULT\_THREAD\_COUNT);

}

break;

default:

setThreadCount(DEFAULT\_THREAD\_COUNT);

}

}

private void setThreadCount(int threadCount) {

setCorePoolSize(threadCount);

setMaximumPoolSize(threadCount);

}

就如上面所说，根据用户使用不同的网络类型分别设置线程的数量，比如当用户使用的是wifi，线程数量将会设置为4个，4G的话设为3个等，这样根据用户的具体情况来设计线程数量是非常人性化的，也是值得我们效仿的。

ok，到此我们重要的Dispatcher对象的构造方法已完全的解析完成了，从上面的解析中我们很清楚的看到了Dispatcher作为中转站存在的意义，几乎所有的线程转换操作都是由Dispatcher来操控的，当然可能还有小伙伴们并不清楚它是怎么运作的，怎么进入子线程的，那是因为我们的讲解还没有进行到进入子线程的步骤而已，下面将会进一步的讲解。

总结下Dispatcher中所包含的重要对象实例：

①：PicassoExecutorService线程池

②：downloader 下载器

③：针对DispatcherThread线程的DispatcherHandler处理器

④：NetworkBroadcastReceiver网络监听器

⑤：mainThreadHandler主线程的Handler

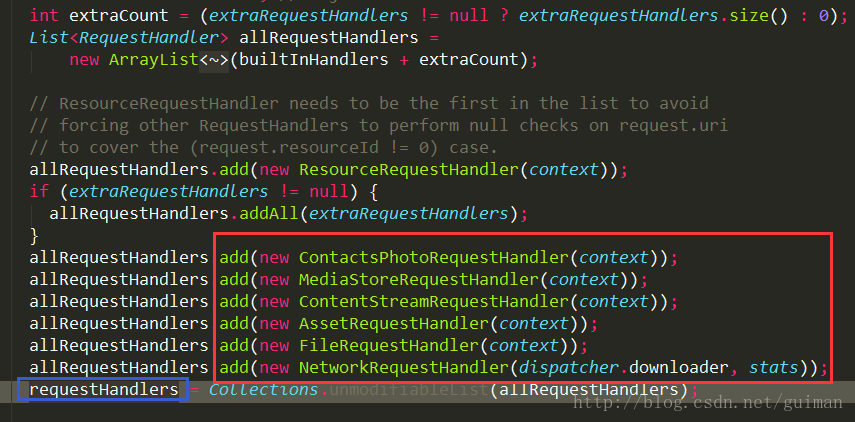
⑥：保存数据的集合：hunterMap，pausedActions，batch等

理解好了上面这些对象存在的意义将对下面的理解有着巨大的好处，请没有完全理解的在仔细的阅读一遍（非常重要）。

ok，知道了Dispatcher包含哪些重要的对象实例之后，让我们再回到Picasso的Builder中，在build方法中最终返回的是一个Picasso的对象实例：

return new Picasso(context, dispatcher, cache, listener, transformer, requestHandlers, stats,defaultBitmapConfig, indicatorsEnabled, loggingEnabled);

* 在Picasso的构造方法中值得我们只要注意的是针对requestHandlers的应用：



这里把Picasso能都应用的RequestHandler都添加到集合中，然后根据具体的需求运用相对应的Handler进行业务的处理。

ok，到此Picasso中with方法中所做的事情已经完完全全的展示在您的面前了，相信您对这一步应用理解的很透彻了。

那么来总结下，Picasso现在拥有了那些重要的对象实例：

①：dispatcher，重要性不言而喻，上面已充分的展示了它的重要性，作为中转站，Picasso是必须要拥有了的。最终目的是让我们的主线程进入子线程中去进行耗时的下载操作。

②：requestHandlers，它的存在是为了选择一种请求处理方式，比如说，下载网络图片需要使用NetworkRequestHandler这个请求器。

③：HANDLER，这是主线程的Handler，用来处理返回结果的，比如说图片下载成功后要更新UI就是通过它来完成的，这会在后面图片下载完成后更新UI时详细讲解

④：一些配置对象实例等，如：缓存cache，图片加载默认配置defaultBitmapConfig，状态存储stats等等。

load

with方法中主要是做了一个基础的配置工作，比如Picasso的配置，Dispatcher的配置，这些都是非常重要的前提工作，只有做好了这些配置我们使用起来才能显得毫不费劲。

下面我们就来看看它的应用吧。在load方法中需要我们传递一个参数，这个参数可以是Url，可以是一个path路径，也可以是一个文件，一个资源布局等：

①：url

public RequestCreator load(Uri uri) {

return new RequestCreator(this, uri, 0);

}

②：path

public RequestCreator load(String path) {

if (path == null) {

return new RequestCreator(this, null, 0);

}

if (path.trim().length() == 0) {

throw new IllegalArgumentException("Path must not be empty.");

}

return load(Uri.parse(path));

}

③：file

public RequestCreator load(File file) {

if (file == null) {

return new RequestCreator(this, null, 0);

}

return load(Uri.fromFile(file));

}

④：resourceId

public RequestCreator load(int resourceId) {

if (resourceId == 0) {

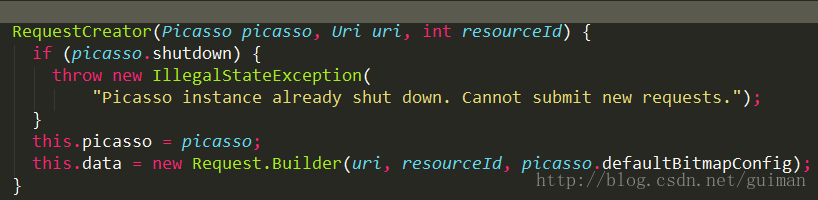
throw new IllegalArgumentException("Resource ID must not be zero.");

}

return new RequestCreator(this, null, resourceId);

}

* 不管传递的是一个什么参数，它都是返回一个请求构造器RequestCreator，我们来看看它的构造方法做了哪些事情：



主要是获取到Picasso对象，并Builder模式构建一个Request中的Builder对象：

Builder(Uri uri, int resourceId, Bitmap.Config bitmapConfig) {

this.uri = uri;

this.resourceId = resourceId;

this.config = bitmapConfig;

}

* 这个Builder对象主要接受了一些参数信息，包括url，资源布局，和默认的图片配置。

由于load方法返回的是一个RequestCreator对象，所以我们可以使用方法链的形式进行调用其他的方法，比如给请求添加占位图，异常图，转换器等等，具体的可以看源码。

ok，总体来说这load方法中主要是创建了一个RequestCreator对象，并且RequestCreator中构造了一个Request.Builder对象。

那么来看看RequestCreator拥有哪些重要的对象实例：

①：picasso对象

②：Request.Builder对象实例data，它里面包括我们请求的URL地址，资源文件以及图片默认配置。

into

在load方法中主要创建了一个RequestCreator对象，并获取到了要加载的url/path/file/resourceId资源地址路径，那么接下来要做的就是在into方法中来加载图片了。先来看看into方法中做了哪些事情：

public void into(ImageView target, Callback callback) {

long started = System.nanoTime();

checkMain();

if (target == null) {

throw new IllegalArgumentException("Target must not be null.");

}

if (!data.hasImage()) {

picasso.cancelRequest(target);

if (setPlaceholder) {

setPlaceholder(target, getPlaceholderDrawable());

}

return;

}

if (deferred) {

if (data.hasSize()) {

throw new IllegalStateException("Fit cannot be used with resize.");

}

int width = target.getWidth();

int height = target.getHeight();

if (width == 0 || height == 0) {

if (setPlaceholder) {

setPlaceholder(target, getPlaceholderDrawable());

}

picasso.defer(target, new DeferredRequestCreator(this, target, callback));

return;

}

data.resize(width, height);

}

Request request = createRequest(started);

String requestKey = createKey(request);

if (shouldReadFromMemoryCache(memoryPolicy)) {

Bitmap bitmap = picasso.quickMemoryCacheCheck(requestKey);

if (bitmap != null) {

picasso.cancelRequest(target);

setBitmap(target, picasso.context, bitmap, MEMORY, noFade, picasso.indicatorsEnabled);

if (picasso.loggingEnabled) {

log(OWNER\_MAIN, VERB\_COMPLETED, request.plainId(), "from " + MEMORY);

}

if (callback != null) {

callback.onSuccess();

}

return;

}

}

if (setPlaceholder) {

setPlaceholder(target, getPlaceholderDrawable());

}

Action action =

new ImageViewAction(picasso, target, request, memoryPolicy, networkPolicy, errorResId,

errorDrawable, requestKey, tag, callback, noFade);

picasso.enqueueAndSubmit(action);

}

* 如上源码可以知道into还是做了很多的事情，下面一一解析：

①：checkMain()：首先检查主否在主线程运行，如果不是在主线程就会抛出一个应该在主线程运行的异常：throw new IllegalStateException(“Method call should happen from the main thread.”);这说明到这一步还是在主线程运行的。

②：data.hasImage()：这里的data是在之前初始化的Request.Builder对象，它里面包含url地址，resourceId和默认配置，这里是判断uri或resourceId是否为空为0，如果是的话就取消imageview的请求：picasso.cancelRequest(target);

③：deferred：延迟，如果需要延迟的话就会得到执行，然后会去获取data中图片的大小，如果没有的话，就得到target的宽高来重新设置要加载的图片的尺寸：data.resize(width, height);

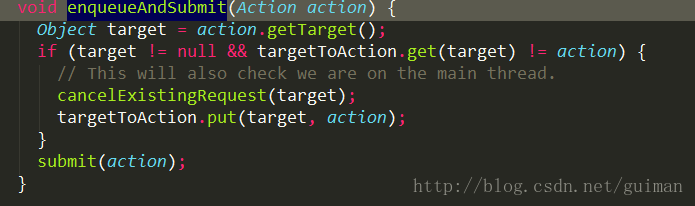
④：createRequest：在data.build()方法中创建Request对象，该对象将会包含uri或resourceId以及默认图片config。然后在得到的Request对象后进行转换，该转换主要是与我们在Picasso构建时是否自定义了RequestTransformer有关。

⑤：createKey：它主要的是返回已String类型key，主要目的是建立ImageView和key的关联，可以通过key来获取到Imageview的状态，比如说是否已缓存

⑥：shouldReadFromMemoryCache：看方法名也能知道，它的作用是是否从内存缓存中读取数据，如果是的话，就从缓存中读取数据，假如获取到数据则会取消当前ImageView的请求并直接给它设置Bitmap，而且如果有回调的话将会回调成功的方法。

⑦：ImageViewAction：构建一个ImageViewAction对象，值得注意的是在构建对象时，会把ImageView添加到RequestWeakReference进行存储，以便于使用时查找，RequestWeakReference是一个WeakReference类型的弱引用。同时ImageViewAction也是在完成图片加载时真正更新UI的关键类，这在后面会进行详细讲解。

⑧：enqueueAndSubmit：调用picasso的enqueueAndSubmit方法进行提交任务。



在来看submit方法的操作：

void submit(Action action) {

dispatcher.dispatchSubmit(action);

}

* 在这里再次使用到了dispatcher任务分发器，把我们的任务action提交到Dispatcher中进行处理。然后在来看下dispatchSubmit方法：

void dispatchSubmit(Action action) {

handler.sendMessage(handler.obtainMessage(REQUEST\_SUBMIT, action));

}

* 由Dispatcher的构造方法我们可以知道此时的handler是DispatcherHandler，那么我们看下它是怎么处理我们的任务行为的：

case REQUEST\_SUBMIT: {

Action action = (Action) msg.obj;

dispatcher.performSubmit(action);

break;

在handleMessage中直接获取到我们的任务行为Action，然后调用performSubmit方法：

void performSubmit(Action action, boolean dismissFailed) {

if (pausedTags.contains(action.getTag())) {

pausedActions.put(action.getTarget(), action);

if (action.getPicasso().loggingEnabled) {

log(OWNER\_DISPATCHER, VERB\_PAUSED, action.request.logId(),

"because tag '" + action.getTag() + "' is paused");

}

return;

}

BitmapHunter hunter = hunterMap.get(action.getKey());

if (hunter != null) {

hunter.attach(action);

return;

}

if (service.isShutdown()) {

if (action.getPicasso().loggingEnabled) {

log(OWNER\_DISPATCHER, VERB\_IGNORED, action.request.logId(), "because shut down");

}

return;

}

hunter = forRequest(action.getPicasso(), this, cache, stats, action);

hunter.future = service.submit(hunter);

hunterMap.put(action.getKey(), hunter);

if (dismissFailed) {

failedActions.remove(action.getTarget());

}

if (action.getPicasso().loggingEnabled) {

log(OWNER\_DISPATCHER, VERB\_ENQUEUED, action.request.logId());

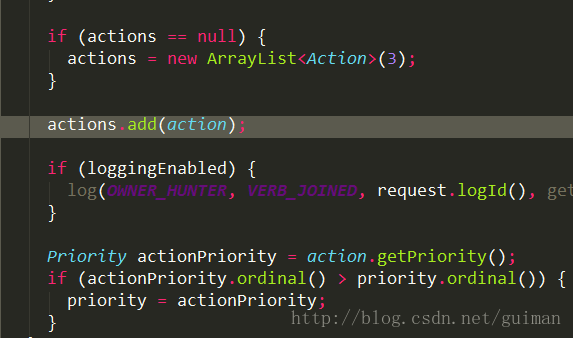
}

}

performSubmit方法是真正意义上的任务提交的具体地方，我们来解读下它的源码：

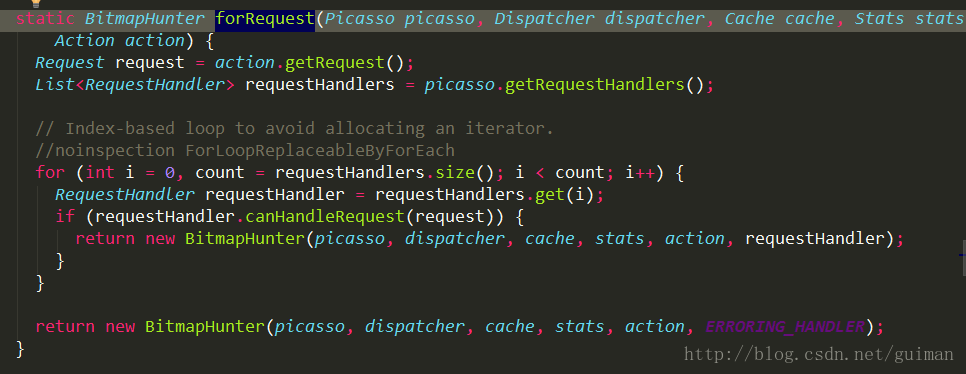
①：首先根据action的标志来查询是否已存在于暂停列表中，如果存在将会把action存放到pausedActions的集合列表中，以便等待唤醒请求。

②：然后通过action的key从hunterMap集合中查询是否已存在该hunterMap的请求项，如果存在将会调用hunter的attach方法，进行合并请求，避免一个ImageView进行多次的重复请求：



把action存放到ArrayList当中，如果有相同的action根据ArrayList的不重复性将会保存一个action，并且更新新的属性值priority。

③：当线程池service没有关闭的时候，通过forRequest方法获取一个Runnable类型的BitmapHunter线程，来看下forRequest的源码：



分别从action和picasso获取到Request请求和所有的requestHandlers请求处理器，然后遍历所有的请求器获取每一个请求处理器，调用canHandleRequest尝试看是否该处理器能够处理，来看下canHandleRequest方法：

public abstract boolean canHandleRequest(Request data);

* 它是一个抽象方法，需要到子类去查找，而实现它的子类都是根据以下的约束条件来判断是否可以处理该请求的：

String scheme = data.uri.getScheme();

* 也就是说通过url地址的Scheme约束条件进行判断的，而以我们现在的action中的Request获取到url，是以http/https开头的，那么来看下NetworkRequestHandler中的canHandleRequest源码：

private static final String SCHEME\_HTTP = "http";

private static final String SCHEME\_HTTPS = "https";

@Override

public boolean canHandleRequest(Request data) {

String scheme = data.uri.getScheme();

return (SCHEME\_HTTP.equals(scheme) || SCHEME\_HTTPS.equals(scheme));

}

可以看出正好的匹配，由此得出结论，将要处理我们请求数据的将是NetworkRequestHandler处理器。

匹配好了请求处理器，将会返回一个BitmapHunter的线程，获取到线程之后会在线程池进行开启线程，并把该线程存放到hunterMap线程集合中以便多次请求可以合并相同的线程：

hunter.future = service.submit(hunter);

hunterMap.put(action.getKey(), hunter);

ok，现在可以到我们的线程池PicassoExecutorService中看看它到底是怎么执行的，submit源码如下：

@Override

public Future<?> submit(Runnable task) {

PicassoFutureTask ftask = new PicassoFutureTask((BitmapHunter) task);

execute(ftask);

return ftask;

}

* 用把我们的线程task封装到PicassoFutureTask 中，PicassoFutureTask 是一个更便于我们控住处理的线程，然后调用execute开启线程，之后会把我们的线程转移到addWorker方法中，在addWorker中开启线程start：

boolean workerStarted = false;

boolean workerAdded = false;

Worker w = null;

try {

w = new Worker(firstTask);

final Thread t = w.thread;

if (t != null) {

final ReentrantLock mainLock = this.mainLock;

mainLock.lock();

try {

// Recheck while holding lock.

// Back out on ThreadFactory failure or if

// shut down before lock acquired.

int rs = runStateOf(ctl.get());

if (rs < SHUTDOWN ||

(rs == SHUTDOWN && firstTask == null)) {

if (t.isAlive()) // precheck that t is startable

throw new IllegalThreadStateException();

workers.add(w);

int s = workers.size();

if (s > largestPoolSize)

largestPoolSize = s;

workerAdded = true;

}

} finally {

mainLock.unlock();

}

if (workerAdded) {

t.start();

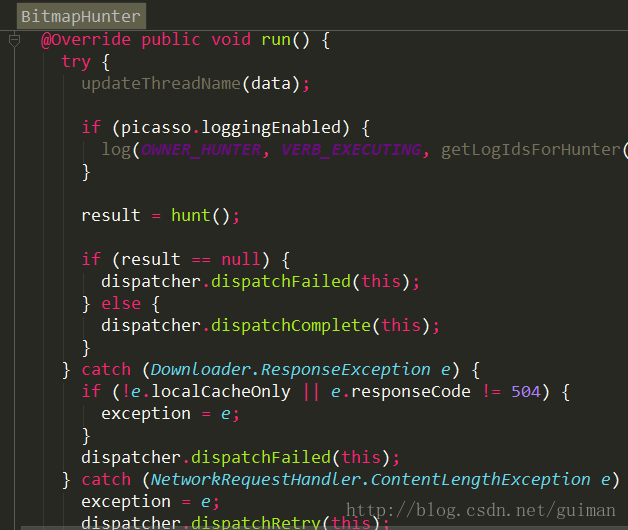
workerStarted = true;

}

上述源码是执行在ThreadPoolExecutor线程池中的，可以看下源码。

当线程开启后，在哪执行呢？

我们知道我们的封装的线程最初始的是BitmapHunter，那么我们就到它里面来看看是怎么执行的：



在BitmapHunter的run方法中，先修改线程名称，然后执行hunt方法，把执行结果存放到result中，那我们先看看hunt方法是怎么执行的：

Bitmap hunt() throws IOException {

Bitmap bitmap = null;

if (shouldReadFromMemoryCache(memoryPolicy)) {

bitmap = cache.get(key);

if (bitmap != null) {

stats.dispatchCacheHit();

loadedFrom = MEMORY;

if (picasso.loggingEnabled) {

log(OWNER\_HUNTER, VERB\_DECODED, data.logId(), "from cache");

}

return bitmap;

}

}

data.networkPolicy = retryCount == 0 ? NetworkPolicy.OFFLINE.index : networkPolicy;

RequestHandler.Result result = requestHandler.load(data, networkPolicy);

if (result != null) {

loadedFrom = result.getLoadedFrom();

exifRotation = result.getExifOrientation();

bitmap = result.getBitmap();

// If there was no Bitmap then we need to decode it from the stream.

if (bitmap == null) {

InputStream is = result.getStream();

try {

bitmap = decodeStream(is, data);

} finally {

Utils.closeQuietly(is);

}

}

}

if (bitmap != null) {

if (picasso.loggingEnabled) {

log(OWNER\_HUNTER, VERB\_DECODED, data.logId());

}

stats.dispatchBitmapDecoded(bitmap);

if (data.needsTransformation() || exifRotation != 0) {

synchronized (DECODE\_LOCK) {

if (data.needsMatrixTransform() || exifRotation != 0) {

bitmap = transformResult(data, bitmap, exifRotation);

if (picasso.loggingEnabled) {

log(OWNER\_HUNTER, VERB\_TRANSFORMED, data.logId());

}

}

if (data.hasCustomTransformations()) {

bitmap = applyCustomTransformations(data.transformations, bitmap);

if (picasso.loggingEnabled) {

log(OWNER\_HUNTER, VERB\_TRANSFORMED, data.logId(), "from custom transformations");

}

}

}

if (bitmap != null) {

stats.dispatchBitmapTransformed(bitmap);

}

}

}

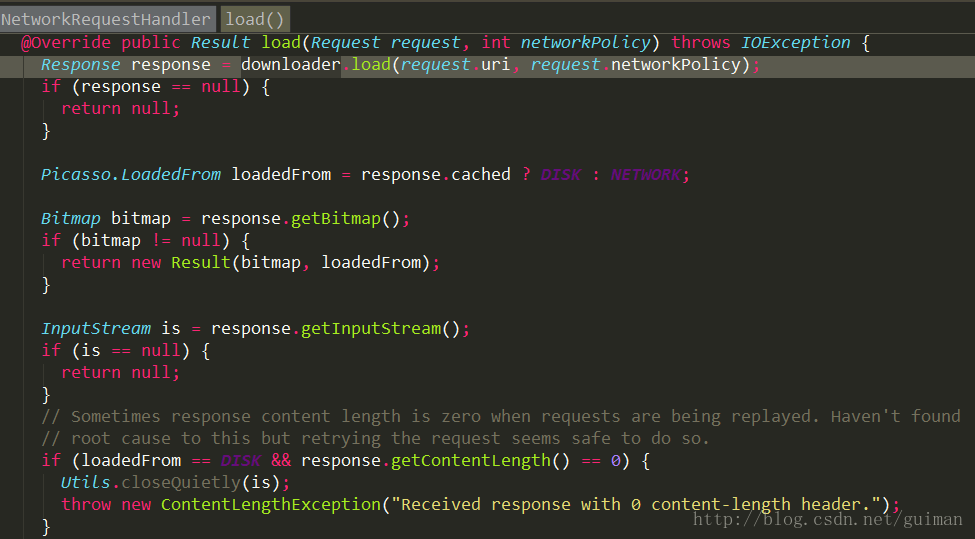
return bitmap;

}

* 解析下源码：

①：还是首先获取到action行为对应key，通过key从缓存cache中查找bitmap是否存在，如果存在修改stats状态并直接的把bitmap返回。

②：如果缓存中不存在则是要去网络加载requestHandler.load();我们通过上面的分析知道能处理当前requesr的requestHandler是NetworkRequestHandler，那么我们去NetworkRequestHandler的load方法中查看：



这里很清晰的可以看到是直接调用downloader的load方法，而我们的downloader在Picasso构建Builder的时候也很清晰的说明是UrlConnectionDownloader，那么在去UrlConnectionDownloader的load方法看看：

protected HttpURLConnection openConnection(Uri path) throws IOException {

HttpURLConnection connection = (HttpURLConnection) new URL(path.toString()).openConnection();

connection.setConnectTimeout(Utils.DEFAULT\_CONNECT\_TIMEOUT\_MILLIS);

connection.setReadTimeout(Utils.DEFAULT\_READ\_TIMEOUT\_MILLIS);

return connection;

}

@Override public Response load(Uri uri, int networkPolicy) throws IOException {

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.ICE\_CREAM\_SANDWICH) {

installCacheIfNeeded(context);

}

HttpURLConnection connection = openConnection(uri);

connection.setUseCaches(true);

if (networkPolicy != 0) {

String headerValue;

if (NetworkPolicy.isOfflineOnly(networkPolicy)) {

headerValue = FORCE\_CACHE;

} else {

StringBuilder builder = CACHE\_HEADER\_BUILDER.get();

builder.setLength(0);

if (!NetworkPolicy.shouldReadFromDiskCache(networkPolicy)) {

builder.append("no-cache");

}

if (!NetworkPolicy.shouldWriteToDiskCache(networkPolicy)) {

if (builder.length() > 0) {

builder.append(',');

}

builder.append("no-store");

}

headerValue = builder.toString();

}

connection.setRequestProperty("Cache-Control", headerValue);

}

int responseCode = connection.getResponseCode();

if (responseCode >= 300) {

connection.disconnect();

throw new ResponseException(responseCode + " " + connection.getResponseMessage(),

networkPolicy, responseCode);

}

long contentLength = connection.getHeaderFieldInt("Content-Length", -1);

boolean fromCache = parseResponseSourceHeader(connection.getHeaderField(RESPONSE\_SOURCE));

return new Response(connection.getInputStream(), fromCache, contentLength);

}

这里的代码相信我们都很熟悉，就是构建一个HttpURLConnection 去进行网络请求，当然还有就是根据networkPolicy进行一些网络缓存的策略。最后把结果存放到Response对象中。

然后NetworkRequestHandler的load方法又会从Response对象中获取数据，并把它存放到Result对象中。然后返回给BitmapHunter中hunt方法的RequestHandler.Result result中，从result中获取输入流，解析输入流转化为Bitmap并返回。

到此我们已从网络中下载了数据，并转化为bitmap了，然后在返回我们的BitmapHunter中的run方法中，在run方法中我们获取到了bitmap，然后调用dispatcher进行事物分发，成功获取则调用dispatchComplete，否则调用dispatchFailed。

下面我们看看dispatcher中的dispatchComplete方法：

void dispatchComplete(BitmapHunter hunter) {

handler.sendMessage(handler.obtainMessage(HUNTER\_COMPLETE, hunter));

}

同样的道理，这里的handler还是DispatcherHandler，那么来看看它的处理：

case HUNTER\_COMPLETE: {

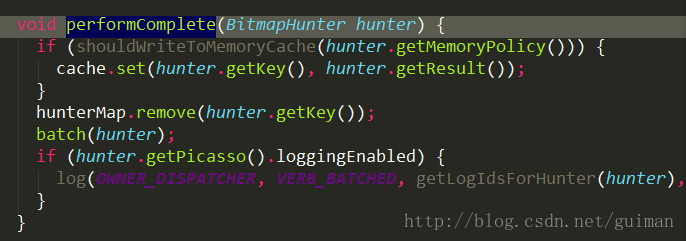
BitmapHunter hunter = (BitmapHunter) msg.obj;

dispatcher.performComplete(hunter);

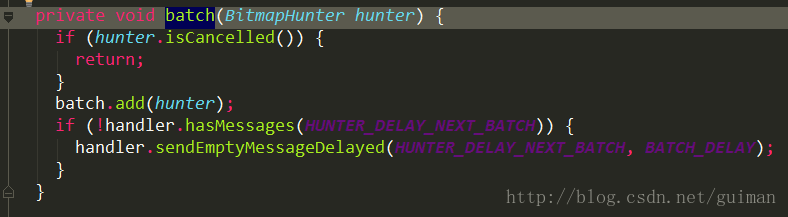
break;

}

转到dispatcher的performComplete方法中：



这里首先把我们的结果保存在cache缓存中，然后从hunterMap集合中移除BitmapHunter对应的key，原因是请求已完成。然后调用 batch(hunter);方法：



首先判断BitmapHunter是否已取消，然后把BitmapHunter存放在一个List< BitmapHunter > batch集合中，最后通过DispatcherHandler发送一个空的延迟消息，目的是为了延迟下下一个网络加载以便处理当前的bitmap工作，来看下它是怎么处理的：

case HUNTER\_DELAY\_NEXT\_BATCH: {

dispatcher.performBatchComplete();

break;

}

进入performBatchComplete中查看：

void performBatchComplete() {

List<BitmapHunter> copy = new ArrayList<BitmapHunter>(batch);

batch.clear();

mainThreadHandler.sendMessage(mainThreadHandler.obtainMessage(HUNTER\_BATCH\_COMPLETE, copy));

logBatch(copy);

}

performBatchComplete中首先获取存放BitmapHunter的集合，然后调用mainThreadHandler发送消息。

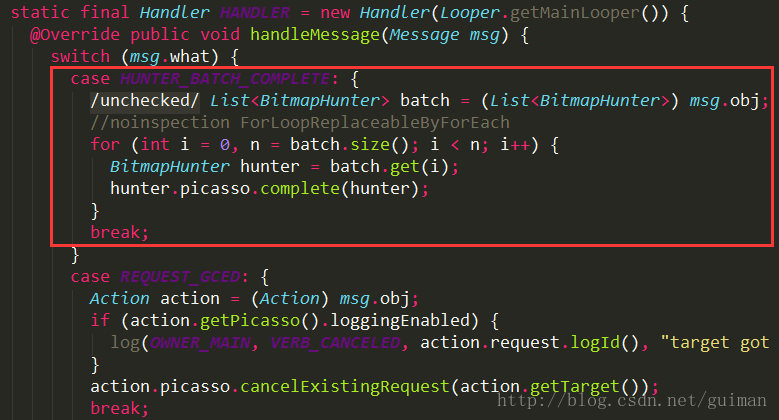
还记得mainThreadHandler是怎么吗？

在Picasso类中构建Builder的build方法中，创建一个Dispatcher对象，里面传进一个HANDLER的参数，请看：

Dispatcher dispatcher = new Dispatcher(context, service, HANDLER, downloader, cache, stats);

这个HANDLER就是mainThreadHandler。

那么它发送的消息是怎么处理的呢？



获取到BitmapHunter集合进行遍历，然后直接调用Picasso中的complete方法：

void complete(BitmapHunter hunter) {

Action single = hunter.getAction();

List<Action> joined = hunter.getActions();

boolean hasMultiple = joined != null && !joined.isEmpty();

boolean shouldDeliver = single != null || hasMultiple;

if (!shouldDeliver) {

return;

}

Uri uri = hunter.getData().uri;

Exception exception = hunter.getException();

Bitmap result = hunter.getResult();

LoadedFrom from = hunter.getLoadedFrom();

if (single != null) {

deliverAction(result, from, single);

}

if (hasMultiple) {

//noinspection ForLoopReplaceableByForEach

for (int i = 0, n = joined.size(); i < n; i++) {

Action join = joined.get(i);

deliverAction(result, from, join);

}

}

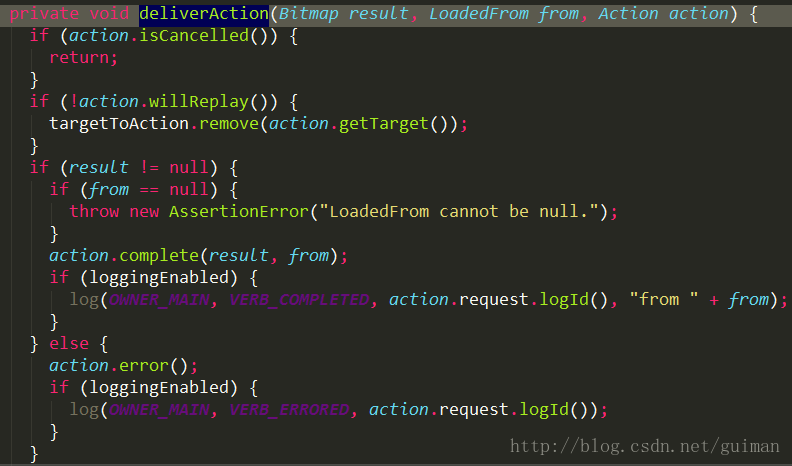
if (listener != null && exception != null) {

listener.onImageLoadFailed(this, uri, exception);

}

}

在complete方法中，直接从BitmapHunter中获取原已封装的Action，Uri，Bitmap等等数据信息，然后调用deliverAction方法：



这里进行一系列的判断，当action没有取消，并且Bitmap不为空时，将会调用action.complete(result, from);来完成操作。

那还记得我们在创建action时的操作吗？

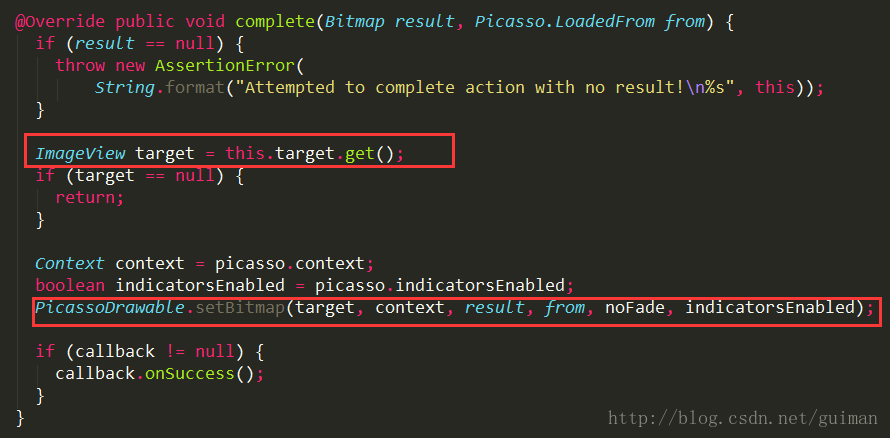
Action action =

new ImageViewAction(picasso, target, request, memoryPolicy, networkPolicy, errorResId,

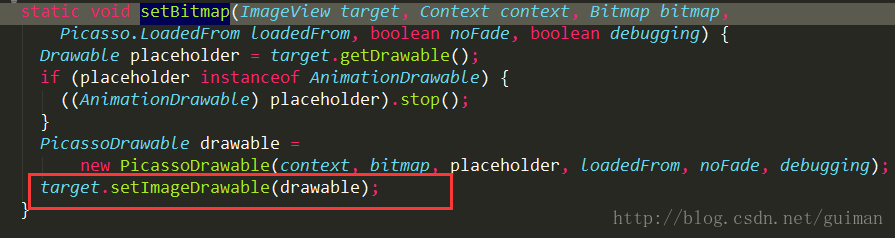
errorDrawable, requestKey, tag, callback, noFade);

那么很清晰的就知道我们的Action 其实就是ImageViewAction呢

那么我们来看下ImageViewAction的complete是怎么操作的呢？



target就是我们在创建ImageViewAction时传递的ImageView，现在获取到它，然后调用PicassoDrawable.setBitmap方法来完成设置图片：



当我们看到target.setImageDrawable(drawable);时，是不是终于松了一口气呢，终于看到了为ImageView设置图片的信息了。

好了，到了这一步整个Picasso加载图片的源码执行流程就已完全的解析完毕了，相信您可以非常清楚的了解了整个框架的加载过程，同时我也相信可能很少人能做到我这么详细的完完全全的分析整个执行过程，而且整个过程中并没有给大家留下什么盲点，是真真正正的源码大解析。

### okhttp的源码解析

#### OkHttp3源码分析[综述]

OkHttp是一个高效的Http客户端，有如下的特点：

1. 支持HTTP2/SPDY黑科技
2. socket自动选择最好路线，并支持自动重连
3. 拥有自动维护的socket连接池，减少握手次数
4. 拥有队列线程池，轻松写并发
5. 拥有Interceptors轻松处理请求与响应（比如透明GZIP压缩,LOGGING）
6. 基于Headers的缓存策略

本文基于okhttp3源码进行分析，逻辑错误或者不足请指出！

建议使用Idea作为分析工具。

## 主要对象

* Connections: 对JDK中的socket进行了引用计数封装，用来控制socket连接
* Streams: 维护HTTP的流，用来对Requset/Response进行IO操作
* Calls: HTTP请求任务封装
* StreamAllocation: 用来控制Connections/Streams的资源分配与释放

## 工作流程的概述

当我们用OkHttpClient.newCall(request)进行execute/enenqueue时，实际是将请求Call放到了Dispatcher中，okhttp使用[Dispatcher](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/Dispatcher.java#L36-L36)进行线程分发，它有两种方法，一个是普通的同步单线程；另一种是使用了队列进行并发任务的分发(Dispatch)与回调，我们下面主要分析第二种，也就是队列这种情况，这也是okhttp能够竞争过其它库的核心功能之一

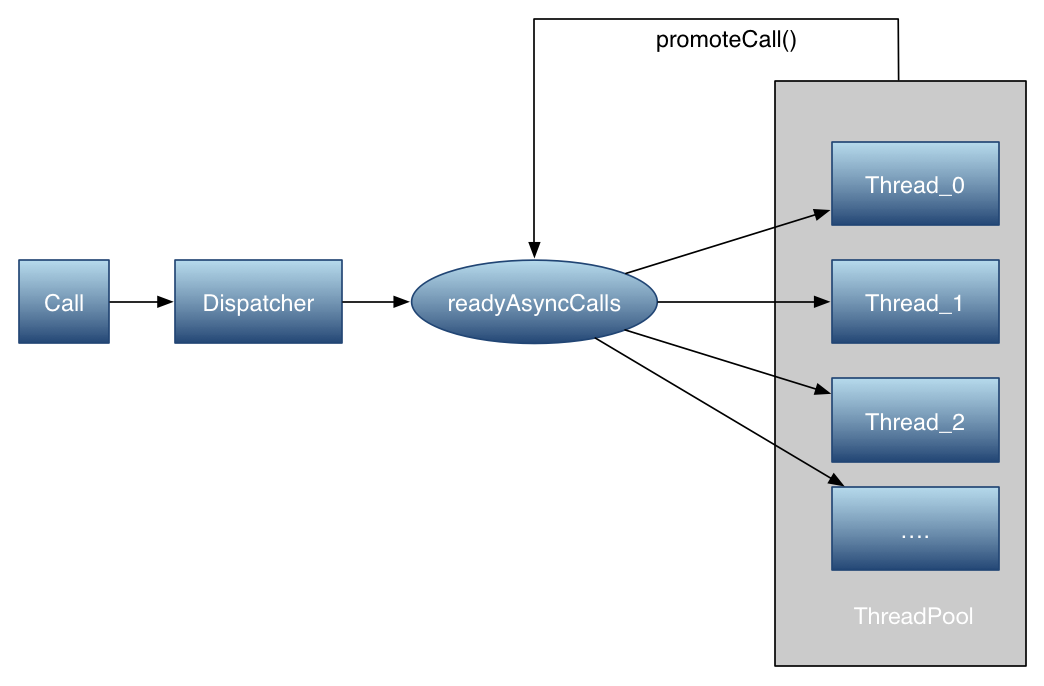
### 1. Dispatcher的结构

Dispatcher维护了如下变量，用于控制并发的请求

* maxRequests = 64: 最大并发请求数为64
* maxRequestsPerHost = 5: 每个主机最大请求数为5
* Dispatcher: 分发者，也就是生产者（默认在主线程）
* AsyncCall: 队列中需要处理的Runnable（包装了异步回调接口）
* ExecutorService：消费者池（也就是线程池）
* Deque<readyAsyncCalls>：缓存（用数组实现，可自动扩容，无大小限制）
* Deque<runningAsyncCalls>：正在运行的任务，仅仅是用来引用正在运行的任务以判断并发量，注意它并不是消费者缓存

根据生产者消费者模型的模型理论，当入队(enqueue)[请求](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L110-L110)时，如果满足(runningRequests<64 && runningRequestsPerHost<5)，那么就直接把AsyncCall直接加到runningCalls的队列中，并在线程池中执行。如果消费者缓存满了，就放入readyAsyncCalls进行缓存等待。

当任务执行完成后,调用[finished](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L142-L142)的promoteCalls()函数，手动移动缓存区（可以看出这里是主动清理的，因此不会发生死锁）



okhttp dispatcher

本部分详细版在[OkHttp3源码分析[任务队列]](http://www.jianshu.com/p/6637369d02e7)

## Socket管理(StreamAllocation)

经过上一步的分配，我们现在需要进行连接了。我们目前有封装好的Request，而进行HTTP连接需要进行Socket握手，Socket握手的前提是根据域名或代理确定Socket的ip与端口。这个环节主要讲了http的握手过程与连接池的管理，分析的对象主要是StreamAllocation

### 1. 选择路线与自动重连(RouteSelector)

此步骤用于获取socket的ip与端口,各位请欣赏[源码](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/RouteSelector.java#L71-L71)中next()的迷之缩进与递归，代码进行了如下事情：

如果Proxy为null:

1. 在构造函数中[设置代理](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/RouteSelector.java#L121)为Proxy.NO\_PROXY
2. 如果缓存中的lastInetSocketAddress为空，就通过DNS（默认是Dns.SYSTEM，包装了jdk自带的lookup函数）[查询](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/RouteSelector.java#L173)，并保存结果，注意结果是数组，即一个域名有多个IP，这就是自动重连的来源
3. 如果还没有查询到就递归调用next查询，直到查到为止
4. 一切next都没有枚举到，抛出NoSuchElementException，退出(这个几乎见不到)

如果Proxy为HTTP:

1. 设置socket的ip为代理地址的ip
2. 设置socket的端口为代理地址的端口
3. 一切next都没有枚举到，抛出NoSuchElementException，退出
4. HTTP代理是不安全的，本文附录有介绍
5. HTTP代理会帮你在远程服务器进行DNS查询
6. 至于socket代理这里就不分析了，它已经不属于应用层了

### 2. 连接socket链路(RealConnection)

当地址，端口准备好了，就可以进行TCP[连接](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/io/RealConnection.java#L87-L87)了（也就是我们常说的TCP三次握手），步骤如下：

1. 如果连接池中已经存在连接，就从中取出(get)RealConnection，如果[没有命中](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L179-L179)就进入下一步
2. 根据选择的路线(Route)，调用Platform.get().connectSocket选择当前平台Runtime下最好的socket库进行握手
3. 将建立成功的RealConnection放入(put)连接池缓存
4. 如果存在TLS，就根据SSL版本与证书进行安全握手
5. 构造HttpStream并维护刚刚的socket连接，管道建立完成

关于Platform,DNS,Proxy详细请看附录

### 3. 释放socket链路(release)

如果不再需要（比如通信完成，连接失败等）此链路后，[释放](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L247-L247)连接（也就是TCP断开的握手）

1. 尝试从缓存的连接池中[删除](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/ConnectionPool.java#L145)(remove)
2. 如果没有命中缓存,就直接调用jdk的socket关闭

本部分详细版见: [OkHttp3源码分析[复用连接池]](http://www.jianshu.com/p/92a61357164b)

## HTTP请求序列化/反序列化

本段主要分析从拼装HTTP套接字到读取的步骤，用[垠神的话](http://www.yinwang.org/blog-cn/2012/08/01/interpreter)说，就是实现了一个Parser。分析的对象是HttpStream接口，在HTTP/1.1下是Http1xStream实现的。

### 1. 获得HTTP流（httpStream）

以下为无缓存，无多次302跳转，网络良好，HTTP/1.1下的GET访问实例分析。

我们已经在上文的RealConnection通过connectSocket()构造HttpStream对象并建立套接字连接（完成三次握手）

httpStream = connect();

在connect()有非常重要的一步，它通过okio库与远程socket建立了I/O连接，为了更好的理解，我们可以把它看成管道

//source 用于获取response

source = Okio.buffer(Okio.source(rawSocket));

//sink 用于write buffer 到server

sink = Okio.buffer(Okio.sink(rawSocket));

Okhttp的I/O使用的是[Okio](https://github.com/square/okio#sources-and-sinks)库，它是java中最好用的I/O API，本人曾经写NFC对这个用的就非常顺手。

* Buffer: Buffer是可变字节，类似于byte[]，相当于传输介质
* source: source是okio库中的输入组件，类似于inputstream，经常在下载中用到。它的重要方法是read(Buffer sink, long byteCount)，从流中读取数据。
* Sink: sink是okio库中的io输出组件，类似于outputstream，经常用于写到file/Socket，它的最重要方法是void write(Buffer source, long byteCount)，写数据到Buffer中

如果把连接看成管道，->为管道的方向，如下图，这里借鉴了go语言的描述

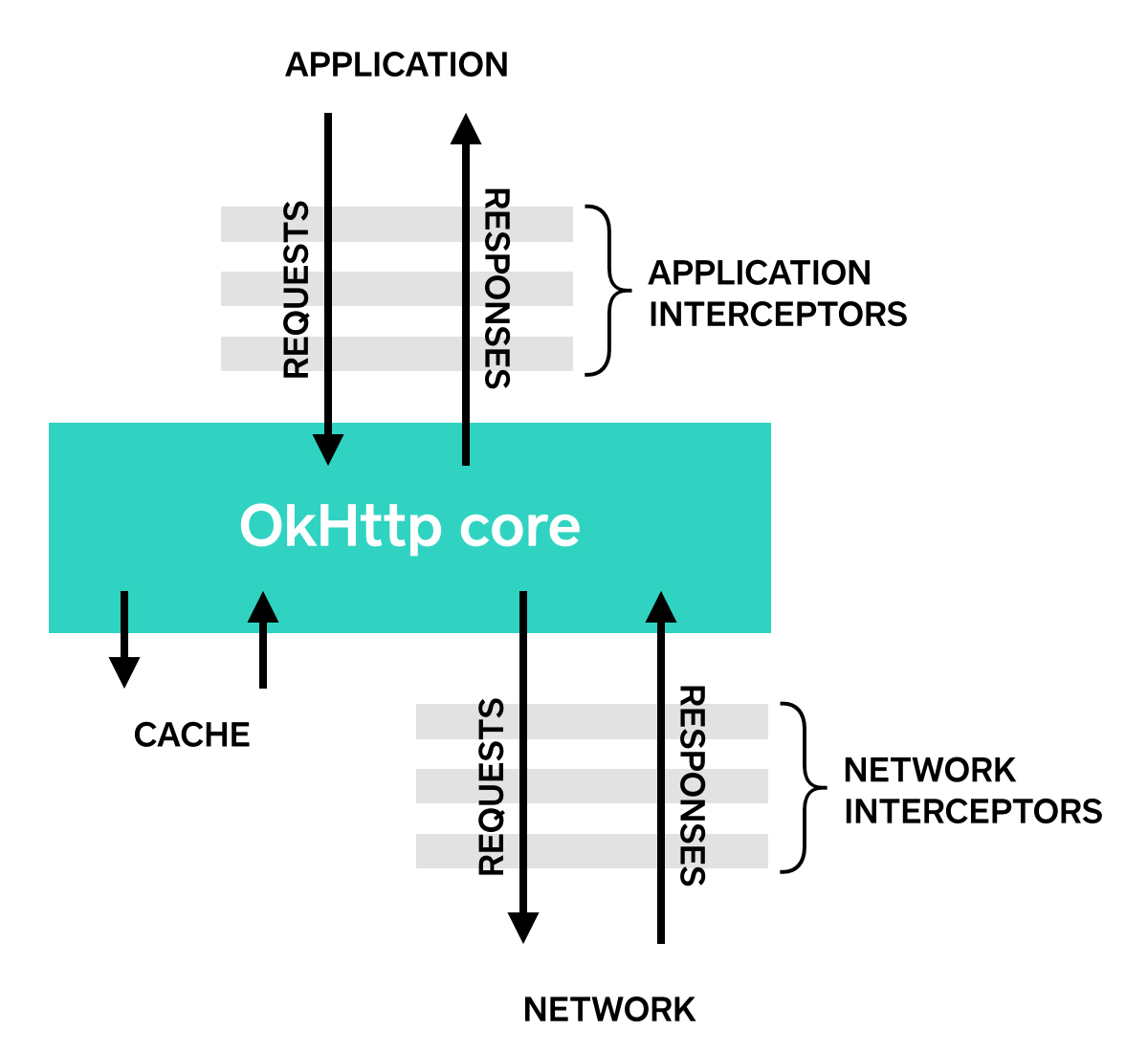
Sink -> Socket/File

Source <- Socket/File

### 2. 拼装Raw请求与Headers(writeRequestHeaders)

我们通过Request.Builder构建了简陋的请求后，可能需要进行一些修饰，这时需要使用Interceptors对Request进行进一步的拼装了。

[拦截器](https://github.com/square/okhttp/wiki/Interceptors)是okhttp中强大的流程装置，它可以用来监控log，修改请求，修改结果，甚至是对用户透明的GZIP压缩。类似于脚本语言中的map操作。在okhttp中，内部维护了一个Interceptors的List，通过InterceptorChain进行多次拦截修改操作。



interceptors

请求的代码如下，详细代码在[这里](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/RealCall.java#L181-L181)，源代码中是自增递归(recursive)调用Chain.process()，直到interceptors().size()中的拦截器全部调用完。这里代码维护性估计看着头大，大神们以后可能把它改成for等更简单的循环，主要做了两件事：

1. 递归调用Interceptors，依次入栈对response进行处理
2. 当全部递归出栈完成后，移交给网络模块(getResponse)

if (index < client.interceptors().size()) {

Interceptor.Chain chain = new ApplicationInterceptorChain(index + 1, request, forWebSocket);

Interceptor interceptor = client.interceptors().get(index);

//递归调用Chain.process()

Response interceptedResponse = interceptor.intercept(chain);

if (interceptedResponse == null) {

throw new NullPointerException("application interceptor " + interceptor

+ " returned null");

}

return interceptedResponse;

}

// No more interceptors. Do HTTP.

return getResponse(request, forWebSocket);

}

接下来是正式的网络请求getResponse()，此步骤通过http协议规范将对象中的数据信息序列化为Raw文本：

1. 在okhttp中，通过RequestLine，Requst，HttpEngine，Header等参数进行序列化操作，也就是拼装参数为socketRaw数据。拼装方法也比较暴力，直接按照RFC协议要求的格式进行concat输出就实现了
2. 通过sink写入write到socket连接。

具体代码在[这里](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/Http1xStream.java#L120)。

### 1.3. 获得响应(readResponseHeaders/Body)

此步骤根据获取到的Socket纯文本，解析为Response对象，我们可以看成是一个反序列化（通过http协议将Raw文本转成对象）的过程：

拦截器的设计:

1. 自定义网络拦截器请求进行递归入栈
2. 在自定义网络拦截器的intercept中，调用NetworkInterceptorChain的proceed(request),进行真正的网络请求(readNetworkResponse)
3. 接自定义请求递归出栈

网络读取(readNetworkResponse)分析:

1. 读取Raw的第一行，并[反序列化](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/Http1xStream.java#L178)为StatusLine对象
2. 以Transfer-Encoding: chunked的模式传输并[组装](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/Http1xStream.java#L407)Body

伪代码如下：

(RawData <- RemoteChannel(www.xx.com, 80))//读取远程的Raw

map(func NetworkInterceptorChains())//预处理

//这里的source引用了HttpEngine，并重写了read方法

.map(func getTransferStream(){})

//根据source拼装body对象

.map(func RealResponseBody(){})

接下来进行释放socket连接，上文已经介绍过了。现在我们就获得到response对象，可以进行进一步的Gson等操作了。

## 附录

以下为一些计算机常识

#### 1. Proxy

代理，也就是有个中间服务器帮助你访问，okhttp中使用jdk自带的代理

You ---- Proxy ----- Server

HTTP代理的本质是改Header信息，当你访问HTTP/HTTPS服务时，本质是明文向跳板发送如下raw，远程服务器帮你完成dns与请求操作，比如HTTPS请求[源码](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/io/RealConnection.java#L234-L234)就详细的解释了发送的内容是非加密的，下面是我实际抓包的内容

//HTTP 请求

GET HTTP://www.qq.com HTTP/1.1

//HTTPS 请求

CONNECT github.com:443 HTTP/1.1

上面的抓包过程，廉价的民用上网行为管理交换机就可以把你记录的一清二楚，所以慎用HTTP代理或者尽量使用HTTPS代理，它是“不安全”的。

#### 2. DNS

DNS也就是域名到ip的映射(mapping)操作，用户向DNS服务器的53端口发送udp包后，会返回域名对应的地址，当然发送udp的细节对用户是透明的，用户直接调用jdk就可以了。我们先试下Unix下的查询

$ host baidu.com

baidu.com has address 111.13.101.208

baidu.com has address 123.125.114.144

.....

在OkHttp中，提供了[DNS接口](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/Dns.java#L31-L31)，默认是使用Dns.SYSTEM，它包装了java原生socket包中的InetAddress.getAllByName(hostname)方法。

参考：[DNSPod中HTTP DNS的实现](http://www.jianshu.com/p/9803a6efb672)

#### 3. Platform

OkHttp的最底层是Socket，而不是URLConnection，它通过Platform的Class.forName()反射获得当前Runtime使用的socket库，调用栈如下（了解即可）

okhttp//实现HTTP协议

framwork//JRE，实现JDK中Socket封装

jvm//JDK的实现，本质对libc标准库的native封装

bionic//android下的libc标准库

systemcall//用户态切换入内核

kernel//实现下协议栈(L4,L3)与网络驱动(一般是L2,L1)

如果你想用蓝牙硬件中Socket的进行HTTP协议开发，尝试重写这个类。

另外，再说一句废话，自从Android4.4以来，URLConnection在fram的实现也是使用了okhttp

OkHttp支持非常多平台下的Socket库实现，包括Android, JettyBootPlatform等都是支持的，具体的平台支持可以看[这里](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/Platform.java#L37)

#### 4. 如何调试HTTP发送的内容

如果需要对OkHttp进行调试，可以看

1. [抓包方法](http://www.jianshu.com/p/9da13b95d231)
2. [okhttp-logging-interceptor](https://github.com/square/okhttp/tree/master/okhttp-logging-interceptor)

#### OkHttp3源码分析[复用连接池]

## 1. 概述

HTTP中的keepalive连接在网络性能优化中，对于延迟降低与速度提升的有非常重要的作用。

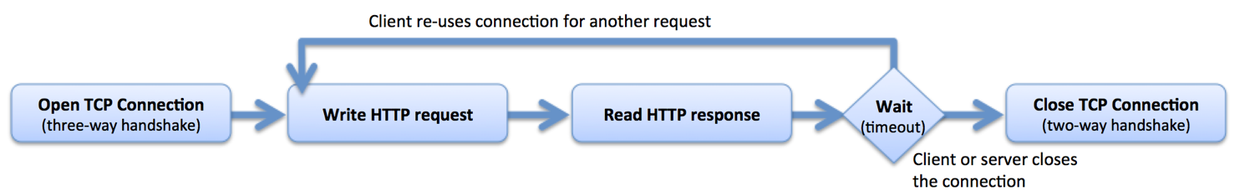
通常我们进行http连接时，首先进行tcp握手，然后传输数据，最后释放



图源: Nginx closed

这种方法的确简单，但是在复杂的网络内容中就不够用了，创建socket需要进行3次握手，而释放socket需要2次握手(或者是4次)。重复的连接与释放tcp连接就像每次仅仅挤1mm的牙膏就合上牙膏盖子接着再打开接着挤一样。而每次连接大概是TTL一次的时间(也就是ping一次)，在TLS环境下消耗的时间就更多了。很明显，当访问复杂网络时，延时（而不是带宽）将成为非常重要的因素。

当然，上面的问题早已经解决了，在http中有一种叫做keepalive connections的机制，它可以在传输数据后仍然保持连接，当客户端需要再次获取数据时，直接使用刚刚空闲下来的连接而不需要再次握手



图源: Nginx keep\_alive

在现代浏览器中，一般同时开启6～8个keepalive connections的socket连接，并保持一定的链路生命，当不需要时再关闭；而在服务器中，一般是由软件根据负载情况(比如FD最大值、Socket内存、超时时间、栈内存、栈数量等)决定是否主动关闭。

Okhttp支持5个并发KeepAlive，默认链路生命为5分钟(链路空闲后，保持存活的时间)

当然keepalive也有缺点，在提高了单个客户端性能的同时，复用却阻碍了其他客户端的链路速度，具体来说如下

1. 根据TCP的拥塞机制，当总水管大小固定时，如果存在大量空闲的keepalive connections（我们可以称作僵尸连接或者泄漏连接），其它客户端们的正常连接速度也会受到影响，这也是运营商为何限制P2P连接数的道理
2. 服务器/防火墙上有并发限制，比如apache服务器对每个请求都开线程，导致只支持150个并发连接（数据来源于nginx官网），不过这个瓶颈随着高并发server软硬件的发展（golang/分布式/IO多路复用）将会越来越少
3. 大量的DDOS产生的僵尸连接可能被用于恶意攻击服务器，耗尽资源

好了，以上科普完毕，本文主要是写客户端的，服务端不再介绍。

下文假设服务器是经过专业的运维配置好的，它默认开启了keep-alive，并不主动关闭连接

## 2. 连接池的使用与分析

首先先说下源码中关键的对象：

* Call: 对http的请求封装，属于程序员能够接触的上层高级代码
* Connection: 对jdk的socket物理连接的包装，它内部有List<WeakReference<StreamAllocation>>的引用
* StreamAllocation: 表示Connection被上层高级代码的引用次数
* ConnectionPool: Socket连接池，对连接缓存进行回收与管理，与CommonPool有类似的设计
* Deque: Deque也就是双端队列，双端队列同时具有队列和栈性质，经常在缓存中被使用，这个是java基础

在okhttp中，连接池对用户，甚至开发者都是透明的。它自动创建连接池，自动进行泄漏连接回收，自动帮你管理线程池，提供了put/get/clear的接口，甚至内部调用都帮你写好了。

在以前的内存泄露[分析文章](http://www.jianshu.com/p/c59c199ca9fa)中我写到，我们知道在socket连接中，也就是Connection中，本质是封装好的流操作，除非手动close掉连接，基本不会被GC掉，非常容易引发内存泄露。所以当涉及到并发socket编程时，我们就会非常紧张，往往写出来的代码都是try/catch/finally的迷之缩进，却又对这样的代码无可奈何。

在okhttp中，在高层代码的调用中，使用了类似于引用计数的方式跟踪Socket流的调用，这里的计数对象是StreamAllocation，它被反复执行[aquire](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L296-L296)与[release](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L301-L301)操作(点击函数可以进入github查看)，这两个函数其实是在改变Connection中的List<WeakReference<StreamAllocation>>大小。List中Allocation的数量也就是物理socket被引用的计数（Refference Count），如果计数为0的话，说明此连接没有被使用，是空闲的，需要通过下文的算法实现回收；如果上层代码仍然引用，就不需要关闭连接。

引用计数法：给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用。它不能处理循环引用的问题。

#### 2.1. 实例化

在源码中，我们先找ConnectionPool实例化的位置，它是直接new出来的，而它的各种操作却在OkHttpClient的[static区](https://github.com/square/okhttp/blob/ee83d8d26afd92d27fbcd2a328e882f25a5090c6/okhttp/src/main/java/okhttp3/OkHttpClient.java#L65-L65)实现了Internal.instance接口作为ConnectionPool的包装。

至于为什么需要这么多此一举的分层包装，主要是为了让外部包的成员访问非public方法，详见这里[注释](https://github.com/square/okhttp/blob/21d63034188d90ca51d635be348d5deba4abeca3/okhttp/src/main/java/okhttp3/internal/Internal.java#L34-L34)

#### 2.2. 构造

1. 连接池内部维护了一个叫做OkHttp ConnectionPool的ThreadPool，专门用来淘汰末位的socket，当满足以下条件时，就会进行末位淘汰，非常像GC
2. 1. 并发socket空闲连接超过5个

2. 某个socket的keepalive时间大于5分钟

1. 维护着一个Deque<Connection>，提供get/put/remove等数据结构的功能
2. 维护着一个RouteDatabase，它用来记录连接失败的Route的黑名单，当连接失败的时候就会把失败的线路加进去（本文不讨论）

#### 2.3 put/get操作

在连接池中，提供如下的操作，这里可以看成是对deque的一个简单的包装

//从连接池中获取

get

//放入连接池

put

//线程变成空闲，并调用清理线程池

connectionBecameIdle

//关闭所有连接

evictAll

随着上述操作被更高级的对象调用，Connection中的StreamAllocation被不断的[aquire](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L296-L296)与[release](https://github.com/square/okhttp/blob/c64e3426a326fdf61a6f9859292a45845186e790/okhttp/src/main/java/okhttp3/internal/http/StreamAllocation.java#L301-L301)，也就是List<WeakReference<StreamAllocation>>的大小将时刻变化

#### 2.4 Connection自动回收的实现

java内部有垃圾回收GC，okhttp有socket的回收；垃圾回收是根据对象的引用树实现的，而okhttp是根据RealConnection的虚引用StreamAllocation引用计数是否为0实现的。我们先看代码

cleanupRunnable:

当用户socket连接成功，向连接池中put新的socket时，回收函数会被主动调用，线程池就会执行cleanupRunnable，如下

//Socket清理的Runnable，每当put操作时，就会被主动调用

//注意put操作是在网络线程

//而Socket清理是在`OkHttp ConnectionPool`线程池中调用

while (true) {

//执行清理并返回下场需要清理的时间

long waitNanos = cleanup(System.nanoTime());

if (waitNanos == -1) return;

if (waitNanos > 0) {

synchronized (ConnectionPool.this) {

try {

//在timeout内释放锁与时间片

ConnectionPool.this.wait(TimeUnit.NANOSECONDS.toMillis(waitNanos));

} catch (InterruptedException ignored) {

}

}

}

}

这段死循环实际上是一个阻塞的清理任务，首先进行清理(clean)，并返回下次需要清理的间隔时间，然后调用wait(timeout)进行等待以释放锁与时间片，当等待时间到了后，再次进行清理，并返回下次要清理的间隔时间...

Cleanup:

[cleanup](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/ConnectionPool.java#L183-L183)使用了类似于GC的标记-清除算法，也就是首先标记出最不活跃的连接(我们可以叫做泄漏连接，或者空闲连接)，接着进行清除，流程如下:

long cleanup(long now) {

int inUseConnectionCount = 0;

int idleConnectionCount = 0;

RealConnection longestIdleConnection = null;

long longestIdleDurationNs = Long.MIN\_VALUE;

//遍历`Deque`中所有的`RealConnection`，标记泄漏的连接

synchronized (this) {

for (RealConnection connection : connections) {

// 查询此连接内部StreamAllocation的引用数量

if (pruneAndGetAllocationCount(connection, now) > 0) {

inUseConnectionCount++;

continue;

}

idleConnectionCount++;

//选择排序法，标记出空闲连接

long idleDurationNs = now - connection.idleAtNanos;

if (idleDurationNs > longestIdleDurationNs) {

longestIdleDurationNs = idleDurationNs;

longestIdleConnection = connection;

}

}

if (longestIdleDurationNs >= this.keepAliveDurationNs

|| idleConnectionCount > this.maxIdleConnections) {

//如果(`空闲socket连接超过5个`

//且`keepalive时间大于5分钟`)

//就将此泄漏连接从`Deque`中移除

connections.remove(longestIdleConnection);

} else if (idleConnectionCount > 0) {

//返回此连接即将到期的时间，供下次清理

//这里依据是在上文`connectionBecameIdle`中设定的计时

return keepAliveDurationNs - longestIdleDurationNs;

} else if (inUseConnectionCount > 0) {

//全部都是活跃的连接，5分钟后再次清理

return keepAliveDurationNs;

} else {

//没有任何连接，跳出循环

cleanupRunning = false;

return -1;

}

}

//关闭连接，返回`0`，也就是立刻再次清理

closeQuietly(longestIdleConnection.socket());

return 0;

}

太长不想看的话，就是如下的流程：

1. 遍历Deque中所有的RealConnection，标记泄漏的连接
2. 如果被标记的连接满足(空闲socket连接超过5个&&keepalive时间大于5分钟)，就将此连接从Deque中移除，并关闭连接，返回0，也就是将要执行wait(0)，提醒立刻再次扫描
3. 如果(目前还可以塞得下5个连接，但是有可能泄漏的连接(即空闲时间即将达到5分钟))，就返回此连接即将到期的剩余时间，供下次清理
4. 如果(全部都是活跃的连接)，就返回默认的keep-alive时间，也就是5分钟后再执行清理
5. 如果(没有任何连接)，就返回-1,跳出清理的死循环

再次注意：这里的“并发”==(“空闲”＋“活跃”)==5，而不是说并发连接就一定是活跃的连接

pruneAndGetAllocationCount:

如何标记并找到最不活跃的连接呢，这里使用了pruneAndGetAllocationCount的[方法](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/ConnectionPool.java#L238-L238)，它主要依据弱引用是否为null而判断这个连接是否泄漏

//类似于引用计数法，如果引用全部为空，返回立刻清理

private int pruneAndGetAllocationCount(RealConnection connection, long now) {

//虚引用列表

List<Reference<StreamAllocation>> references = connection.allocations;

//遍历弱引用列表

for (int i = 0; i < references.size(); ) {

Reference<StreamAllocation> reference = references.get(i);

//如果正在被使用，跳过，接着循环

//是否置空是在上文`connectionBecameIdle`的`release`控制的

if (reference.get() != null) {

//非常明显的引用计数

i++;

continue;

}

//否则移除引用

references.remove(i);

connection.noNewStreams = true;

//如果所有分配的流均没了，标记为已经距离现在空闲了5分钟

if (references.isEmpty()) {

connection.idleAtNanos = now - keepAliveDurationNs;

return 0;

}

}

return references.size();

}

1. 遍历RealConnection连接中的StreamAllocationList，它维护着一个弱引用列表
2. 查看此StreamAllocation是否为空(它是在线程池的put/remove手动控制的)，如果为空，说明已经没有代码引用这个对象了，需要在List中删除
3. 遍历结束，如果List中维护的StreamAllocation删空了，就返回0，表示这个连接已经没有代码引用了，是泄漏的连接;否则返回非0的值，表示这个仍然被引用，是活跃的连接。

上述实现的过于保守，实际上用filter就可以大致实现，伪代码如下

return references.stream().filter(reference -> {

return !reference.get() == null;

}).count();

## 总结

通过上面的分析，我们可以总结，okhttp使用了类似于引用计数法与标记擦除法的混合使用，当连接空闲或者释放时，StreamAllocation的数量会渐渐变成0，从而被线程池监测到并回收，这样就可以保持多个健康的keep-alive连接，Okhttp的黑科技就是这样实现的。

#### OkHttp3源码分析[缓存策略]

## HTTP缓存基础知识

在分析源码之前，我们先回顾一下http的缓存Header的含义

##### 1. Expires

表示到期时间，一般用在response报文中，当超过此事件后响应将被认为是无效的而需要网络连接，反之而是直接使用缓存

Expires: Thu, 12 Jan 2017 11:01:33 GMT

##### 2. Cache-Control

相对值，单位是秒，指定某个文件被续多少秒的时间，从而避免额外的网络请求。比expired更好的选择，它不用要求服务器与客户端的时间同步，也不用服务器时刻同步修改配置Expired中的绝对时间，而且它的优先级比Expires更高。比如简书静态资源有如下的header，表示可以续31536000秒，也就是一年。

Cache-Control: max-age=31536000, public

##### 3. 修订文件名(Reving Filenames)

如果我们通过设置header保证了客户端可以缓存的，而此时远程服务器更新了文件如何解决呢？我们这时可以通过修改url中的文件名版本后缀进行缓存，比如下文是又拍云的公共CDN就提供了多个版本的JQuery

upcdn.b0.upaiyun.com/libs/jquery/jquery-2.0.3.min.js

##### 4. 条件GET请求(Conditional GET Requests)与304

如缓存果过期或者强制放弃缓存，在此情况下，缓存策略全部交给服务器判断，客户端只用发送条件get请求即可，如果缓存是有效的，则返回304 Not Modifiled，否则直接返回body。

请求的方式有两种：

###### 4.1. Last-Modified-Date:

客户端第一次网络请求时，服务器返回了

Last-Modified: Tue, 12 Jan 2016 09:31:27 GMT

客户端再次请求时，通过发送

If-Modified-Since: Tue, 12 Jan 2016 09:31:27 GMT

交给服务器进行判断，如果仍然可以缓存使用，服务器就返回304

###### 4.2. ETag

ETag是对资源文件的一种摘要，客户端并不需要了解实现细节。当客户端第一请求时，服务器返回了

ETag: "5694c7ef-24dc"

客户端再次请求时，通过发送

If-None-Match:"5694c7ef-24dc"

交给服务器进行判断，如果仍然可以缓存使用，服务器就返回304

如果 ETag 和 Last-Modified 都有，则必须一次性都发给服务器，它们没有优先级之分，反正这里客户端没有任何判断的逻辑。

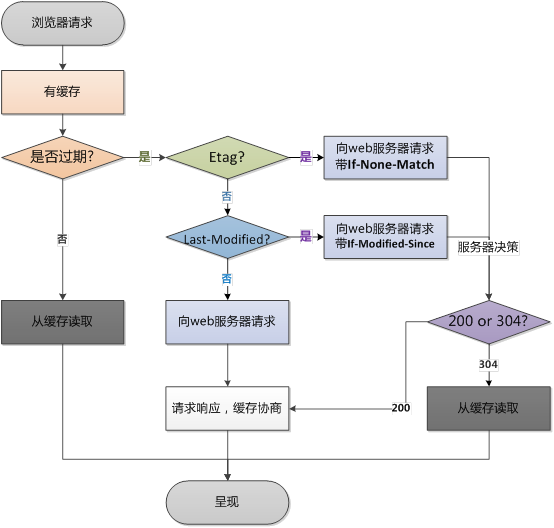
##### 5. 其它标签

* no-cache/no-store: 不使用缓存，no-cache指令的目的是防止从缓存中返回过期的资源。客户端发送的请求中如果包含no-cache指令的话，表示客户端将不会接受缓存过的相应，于是缓存服务器必须把客户端请求转发给源服务器。服务器端返回的相应中包含no-cache指令的话那么缓存服务器不能对资源进行缓存。
* only-if-cached: 只使用缓存
* Date: The date and time that the message was sent
* Age: The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server. 说人话就是CDN反代服务器到原始服务器获取数据延时的缓存时间

"only-if-cached"标签非常具有诱导性，它[只在](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9)请求中使用，表示无论是否有网完全只使用缓存（如果命中还好说，否则返回503错误/网络错误），这个标签比较危险。

全部的标签，可以到[这里看](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields)

以上内容是作为一个服务器开发或者客户端的常识，下图是网上找的总结，注意图中的 ETag 和 Last-Modified 可能有优先级的歧义，你只需要记住它们是没有优先级的。



图源: 浏览器缓存机制 - [吴秦（Tyler）](http://www.cnblogs.com/skynet/)

## 2. 源码分析

OkHttp中使用了[CacheStrategy](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/CacheStrategy.java#L31-L31)实现了上文的流程图，它根据之前的缓存结果与当前将要发送Request的header进行策略分析，并得出是否进行请求的结论。

### 2.1. 总体请求流程分析

CacheStrategy类似一个mapping操作，将两个值输入，再将两个值输出

| **Input** | **request, cacheCandidate** |
| --- | --- |
| ↓ | ↓ |
| CacheStrategy | 处理，判断Header信息 |
| ↓ | ↓ |
| Output | networkRequest, cacheResponse |

Request:  
开发者手动编写并在Interceptor中递归加工而成的对象（如果读者需要调试分析的话，可以用[logging-interceptor](https://github.com/square/okhttp/tree/master/okhttp-logging-interceptor)进行log操作），我们只需要知道了目前传入的Request中并没有任何关于缓存的Header

cacheCandidate:  
也就是上次与服务器交互缓存的Response，可能为null。这里的缓存全部是基于文件系统的Map，key是请求中url的md5，value是在文件中查询到的缓存，页面置换基于[LRU算法](https://en.wikipedia.org/wiki/Page_replacement_algorithm#Least_recently_used)，我们现在只需要知道它是一个可以读取缓存Header的Response即可。

当被CacheStrategy加工输出后，输出networkRequest与cacheResponse，根据是否为空执行不同的请求

| **networkRequest** | **cacheResponse** | **result** |
| --- | --- | --- |
| null | null | only-if-cached(表明不进行网络请求，且缓存不存在或者过期，一定会返回503错误) |
| null | non-null | 不进行网络请求，而且缓存可以使用，直接返回缓存，不用请求网络 |
| non-null | null | 需要进行网络请求，而且缓存不存在或者过期，直接访问网络 |
| non-null | non-null | Header中含有ETag/Last-Modified标签，需要在条件请求下使用，还是需要访问网络 |

以上是对networkRequest/cacheResponse进行findusage查询获得出的结论

基本上与上文的图片完全一致，以上就是OkHttp的缓存策略

关于此部分的分析，读者可以在[HttpEngine](https://github.com/square/okhttp/blob/b8c5938ed49502fbae89d0e842389853a208f996/okhttp/src/main/java/okhttp3/internal/http/HttpEngine.java#L82-L82)对象中通过对userResponse进行findUsage分析得出，源码都是一大堆的if判断

### 2.2. CacheStrategy的加工过程

CacheStrategy使用[Factory模式](https://en.wikipedia.org/wiki/Factory_method_pattern)进行构造，参数如下

InternalCache responseCache = Internal.instance.internalCache(client);

//cacheCandidate从disklurcache中获取

//request的url被md5序列化为key,进行缓存查询

Response cacheCandidate = responseCache != null ? responseCache.get(request) : null;

//请求与缓存

factory = new CacheStrategy.Factory(now, request, cacheCandidate);

cacheStrategy = factory.get();

//输出结果

networkRequest = cacheStrategy.networkRequest;

cacheResponse = cacheStrategy.cacheResponse;

//进行一大堆的if判断，内容同上表格

.....

可以看出Factory.get()是最关键的缓存策略的判断，我们点入get()[方法](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/http/CacheStrategy.java#L166-L166)，可以发现是对getCandidate()的一个封装，我们接着点开getCandidate()，全是if与数学计算，详细代码如下

private CacheStrategy getCandidate() {

//如果缓存没有命中(即null),网络请求也不需要加缓存Header了

if (cacheResponse == null) {

//`没有缓存的网络请求,查上文的表可知是直接访问

return new CacheStrategy(request, null);

}

// 如果缓存的TLS握手信息丢失,返回进行直接连接

if (request.isHttps() && cacheResponse.handshake() == null) {

//直接访问

return new CacheStrategy(request, null);

}

//检测response的状态码,Expired时间,是否有no-cache标签

if (!isCacheable(cacheResponse, request)) {

//直接访问

return new CacheStrategy(request, null);

}

CacheControl requestCaching = request.cacheControl();

//如果请求报文使用了`no-cache`标签(这个只可能是开发者故意添加的)

//或者有ETag/Since标签(也就是条件GET请求)

if (requestCaching.noCache() || hasConditions(request)) {

//直接连接,把缓存判断交给服务器

return new CacheStrategy(request, null);

}

//根据RFC协议计算

//计算当前age的时间戳

//now - sent + age (s)

long ageMillis = cacheResponseAge();

//大部分情况服务器设置为max-age

long freshMillis = computeFreshnessLifetime();

if (requestCaching.maxAgeSeconds() != -1) {

//大部分情况下是取max-age

freshMillis = Math.min(freshMillis, SECONDS.toMillis(requestCaching.maxAgeSeconds()));

}

long minFreshMillis = 0;

if (requestCaching.minFreshSeconds() != -1) {

//大部分情况下设置是0

minFreshMillis = SECONDS.toMillis(requestCaching.minFreshSeconds());

}

long maxStaleMillis = 0;

//ParseHeader中的缓存控制信息

CacheControl responseCaching = cacheResponse.cacheControl();

if (!responseCaching.mustRevalidate() && requestCaching.maxStaleSeconds() != -1) {

//设置最大过期时间,一般设置为0

maxStaleMillis = SECONDS.toMillis(requestCaching.maxStaleSeconds());

}

//缓存在过期时间内,可以使用

//大部分情况下是进行如下判断

//now - sent + age + 0 < max-age + 0

if (!responseCaching.noCache() && ageMillis + minFreshMillis < freshMillis + maxStaleMillis) {

//返回上次的缓存

Response.Builder builder = cacheResponse.newBuilder();

return new CacheStrategy(null, builder.build());

}

//缓存失效, 如果有etag等信息

//进行发送`conditional`请求,交给服务器处理

Request.Builder conditionalRequestBuilder = request.newBuilder();

if (etag != null) {

conditionalRequestBuilder.header("If-None-Match", etag);

} else if (lastModified != null) {

conditionalRequestBuilder.header("If-Modified-Since", lastModifiedString);

} else if (servedDate != null) {

conditionalRequestBuilder.header("If-Modified-Since", servedDateString);

}

//下面请求实质还说网络请求

Request conditionalRequest = conditionalRequestBuilder.build();

return hasConditions(conditionalRequest) ? new CacheStrategy(conditionalRequest,

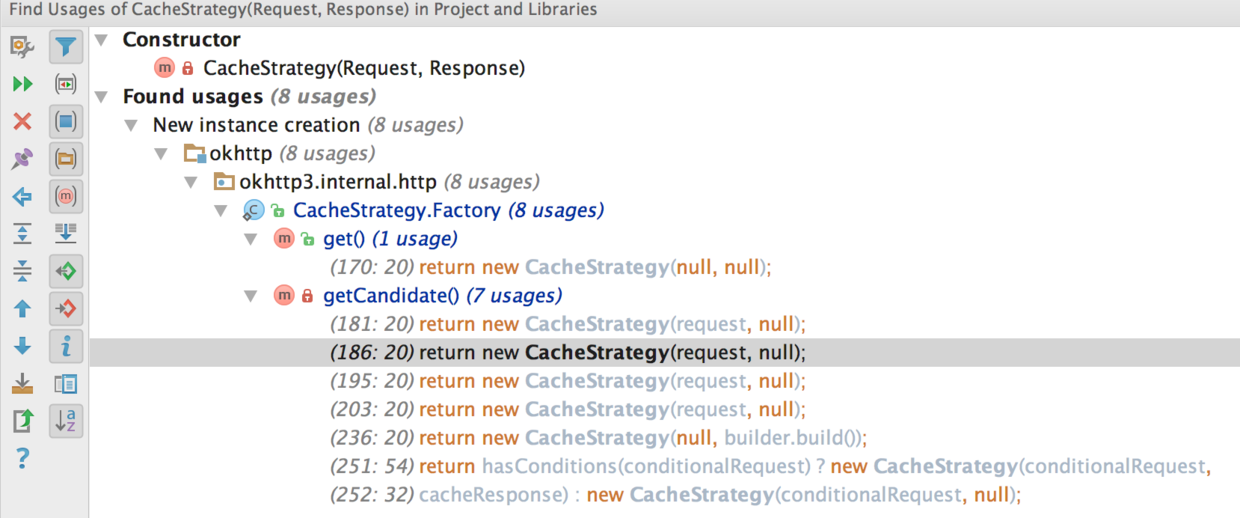
cacheResponse) : new CacheStrategy(conditionalRequest, null);

}

太长不看的话，大多数常见的情况可以用这个估算

now - sent + age < max-age

这里有个技巧，对构造函数进行findUsage查询，就可以看出各个输出是否为空的结果，然后各个击破分析

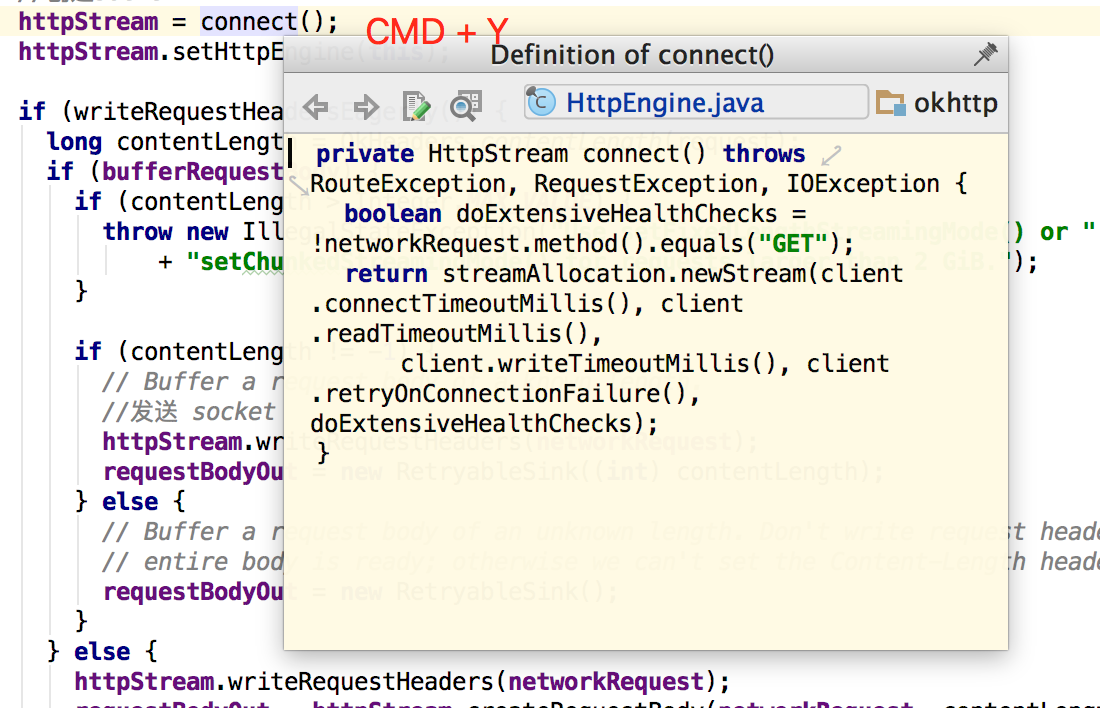


new CacheStrategy()

## 3. 结论

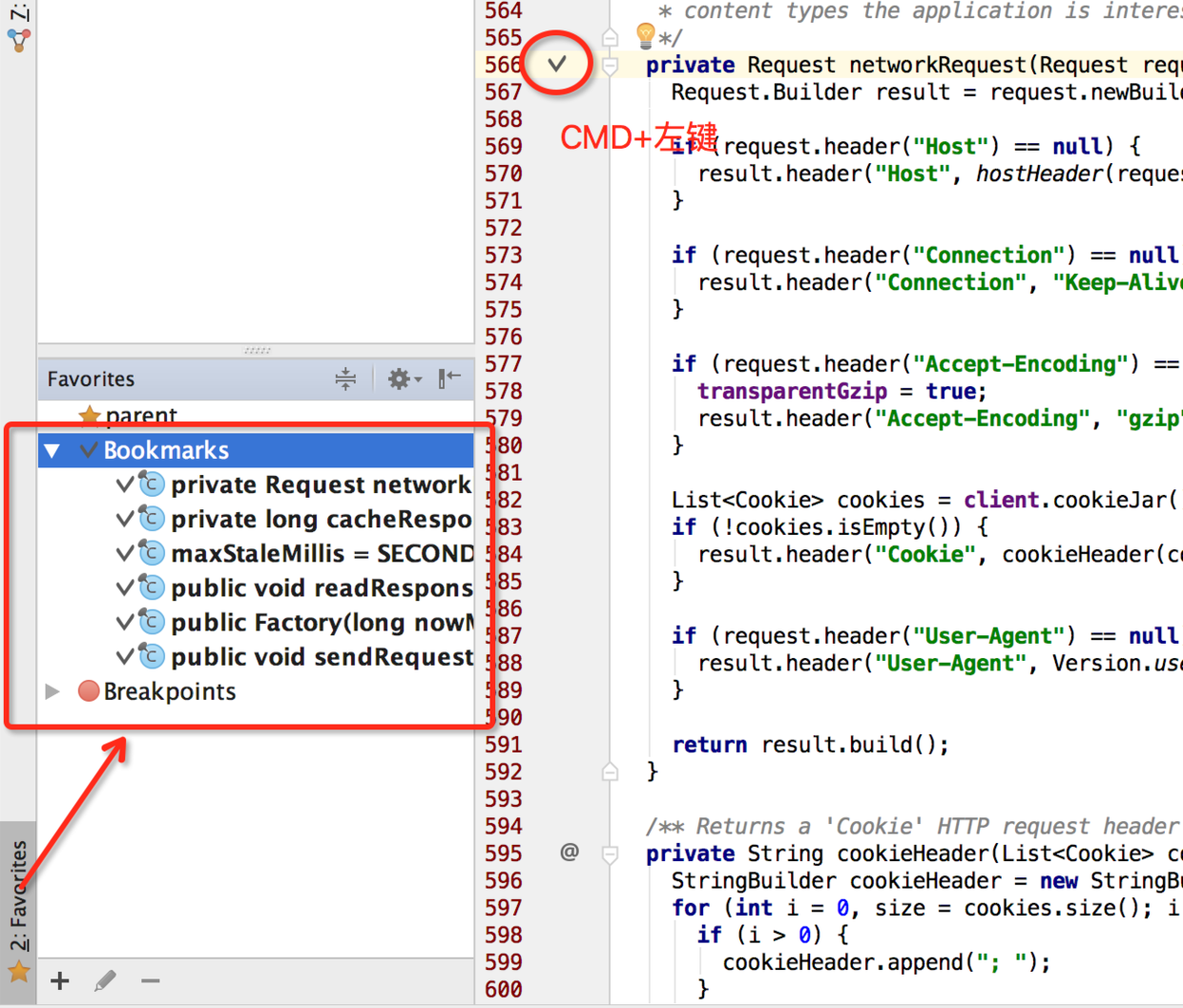
通过上面的分析，我们可以发现，okhttp实现的缓存策略实质上就是大量的if判断集合，这些是根据RFC标准文档写死的，并没有相当难的技巧。

1. Okhttp的缓存是自动完成的，完全由服务器Header决定的，自己没有必要进行控制。网上热传的文章在Interceptor中手工添加缓存代码控制，它固然有用，但是属于Hack式的利用，违反了RFC文档标准，不建议使用，OkHttp的官方缓存控制在[注释中](https://github.com/square/okhttp/blob/d662c1a82851800c46ad8ede2d9d10d10427fdad/okhttp/src/main/java/okhttp3/Cache.java#L79)。如果读者的需求是对象持久化，建议用文件储存或者数据库即可（比如realm）。
2. 服务器的配置非常重要，如果你需要减小请求次数，建议直接找对接人员对max-age等头文件进行优化；服务器的时钟需要严格NTP同步
3. 充分利用Idea的findUsage的功能，源码的各个跳转条件可以很快分析完成
4. 使用CMD + Y可以快速预览某个函数，类似于forcetouch功能



Idea quick preview

1. 使用CMD + 左键可以添加标签，方便跳转代码，如图



#### OkHttp3源码分析[DiskLruCache]

本文目录：

* Cache的简介
* LinkedHashMap原理
* OkHttp的文件系统

本文主要是对put/get过程进行分析，注意缓存的判断依据不是本文，而是[缓存策略](http://www.jianshu.com/p/9cebbbd0eeab)

## 1. Cache的简介

缓存，顾名思义，也就是方便用户快速的获取值的一种储存方式。小到与CPU同频的昂贵的缓存颗粒，内存，硬盘，网络，CDN反代缓存，DNS递归查询，OS页面置换，Redis数据库，都可以看作缓存。它有如下的特点：

1. 缓存载体与持久载体总是相对的，容量远远小于持久容量，成本高于持久容量，速度高于持久容量。比如硬盘与网络，目前主流的SSD硬盘可以达到500MB/S，而很多地区网速却只有4M，将网络中的文件存到硬盘中，硬盘就相当于缓存；再比如内存与硬盘，主流的DDR3内存的速度可以达到10GB/S，而硬盘相对的慢了很多数量级别，将硬盘的游戏加载到内存，内存就相对于硬盘是一种缓存。
2. 需要实现排序依据，在java中，可以使用Comparable<T>作为排序的的接口
3. 需要一种页面置换算法(page replacement algorithm)将旧页面去掉换成新的页面，如最久未使用算法（LFU）、先进先出算法（FIFO）、最近最少使用算法（LFU）、非最近使用算法（NMRU）等
4. 如果没有命中缓存，就需要从原始地址获取，这个步骤叫做“回源”，CDN厂商会标注“回源率”作为卖点

在OkHttp中，使用FileSystem作为缓存载体（磁盘相对于网络的缓存），使用LRU作为页面置换算法（封装了LinkedHashMap）。

1. Comparable<T>是java用来排序的接口，推荐参考阅读《Java Software Structures Designing and Using Data Structures》
2. 页面置换算法可以参考阅读《现代操作系统》的中译本

## 2. LinkedHashMap原理

### 2.1. 源码概述分析

在学习之前，我们要了解一下LinkedHashMap。LinkedHashMap继承于HashMap。

在HashMap中，维护了一个Node<K,V>[] table，当put操作时，将元素按照计算出的Hash填到数组相应位置table[Hash]中，最后迭代时，从table[0]开始向后迭代，具体的顺序取决于元素的HashCode，所以我们常说HashMap的元素迭代是不可预测的。

而在LinkedHashMap中，除了Node<K,V>[] table，还维护着Entry<K,V> head,tail。当put元素后，调用下列回调函数对链表将元素移动到链尾以及清理旧的元素

// move node to last

void afterNodeAccess(Node<K,V> e)

// possibly remove eldest

void afterNodeInsertion(boolean evict)

在get元素时，如果设置accessOrder为true时，通过调用如下回调移动元素到链尾，这里特别强调移动，如果这个元素本身已经在链表中，那它将只会移动，而不是新建

// move node to last

void afterNodeAccess(Node<K,V> e)

综上，当你反复对元素进行get/put操作时，经常使用的元素会被移动到tail中，而长期不用的元素会被移动到head

最后迭代(Iterator)时，迭代是从旧元素迭代到新元素，这就是LRU的实现

head <--> .... <--> tail

旧元素 <-----------> 反复使用的新元素

在OkHttp中，使用了DiskLruCache对LinkedHashMap进行了封装实现LRU，按照下图的方法进行初始化

//按照访问顺序排序的Map，设置accessOrder为true

map = new LinkedHashMap<>(0, 0.75f, true);

### 2.2. HashMap的对比

以下是常见的3种map的区别，以下均不计算扩容时的时间复杂度

|  | **HashMap** | **LinkedHashMap** | **TreeMap** |
| --- | --- | --- | --- |
| Performance get/set | O(1) | O(1) | O(logN) |
| Implement | Array | Link + Array | Red-Black Tree |
| Iteration | unpredictable | put/accessOrder | Comparable<Key> |

上述具体代码没有源码分析哦，王垠大神看了[都会头大](http://www.yinwang.org/blog-cn/2013/03/15/language-design-mistake1/)

1. 需要复习HashMap源码？可以考虑阅读[HashMap原理文章](http://www.jianshu.com/p/e54047b2b563)
2. 本部分基于JDK1.8.0\_05，可能部分函数与网上文章相冲突
3. 在golang中，使用ring与map实现了Lru，可以看[这里](https://github.com/golang/groupcache/blob/master/lru/lru.go)

## 3. OkHttp的文件系统

OkHttp中的关键对象如下：

* FileSystem: 使用Okio对File的封装，简化了IO操作
* DiskLruCache.Editor: 添加了同步锁，并对FileSystem进行高度封装
* DiskLruCache.Entry: 维护着key对应的多个文件
* Cache.Entry: Responsejava对象与Okio流的序列化/反序列化类
* DiskLruCache: 维护着文件的创建，清理，读取。内部有清理线程池，LinkedHashMap(也就是LruCache)
* Cache: 被上级代码调用，提供透明的put/get操作，封装了缓存检查条件与DiskLruCache，开发者只用配置大小即可，不需要手动管理
* Response/Requset: OkHttp的请求与回应

### 3.1. 文件初级封装(FileSystem)

众所周之，文件读写是流操作，是一大堆的令人头痛的try/cache操作，在OkHttp中设计了[FileSystem.SYSTEM](https://github.com/square/okhttp/blob/db9c2db40b0b89a1853715fd52e2748463d9cc9c/okhttp/src/main/java/okhttp3/internal/io/FileSystem.java#L43-L43)作为文件层的管理。通过用Okio库中的Source/Sink对File进行包装，而不用更为头痛的InputStream这类东西，使上层调用与管道操作一样简单。

File(低级操作，步骤繁琐) -> Okio(封装) －> FileSystem(友好工具类)

至于Okio为何这个好，直接去[官网](https://github.com/square/okio)参考

### 3.2. 文件高级封装(DiskLruCache.Entry/Editor/Snapshot)

本部分进行了如下的转换，进行了实际的put/get操作

FileSystem <-- DiskLruCache.Entry/Editor --> source/sink(更少参数)

[DiskLruCache.Entry](https://github.com/square/okhttp/blob/d662c1a82851800c46ad8ede2d9d10d10427fdad/okhttp/src/main/java/okhttp3/internal/DiskLruCache.java#L915)针对每个请求的url对应的文件进行引用维护（而没有进行创建/读取等操作），它内部维护了2个File数组，一般来说每个url对应2～4个文件。 文件名命名规则是{md5(url)+ {0,1}}，后面的0或1，分别表示[ENTRY\_METADATA](https://github.com/square/okhttp/blob/d662c1a82851800c46ad8ede2d9d10d10427fdad/okhttp/src/main/java/okhttp3/Cache.java#L138)与[ENTRY\_BODY]((https://github.com/square/okhttp/blob/d662c1a82851800c46ad8ede2d9d10d10427fdad/okhttp/src/main/java/okhttp3/Cache.java#L138-L138))。

比如在缓存的路径下执行ls，结果如下

$ ls

5716ab0f06c49bc7cf602397c51d5677.0

5716ab0f06c49bc7cf602397c51d5677.1

5b2f52377611dc6201a1871bdb997466.0

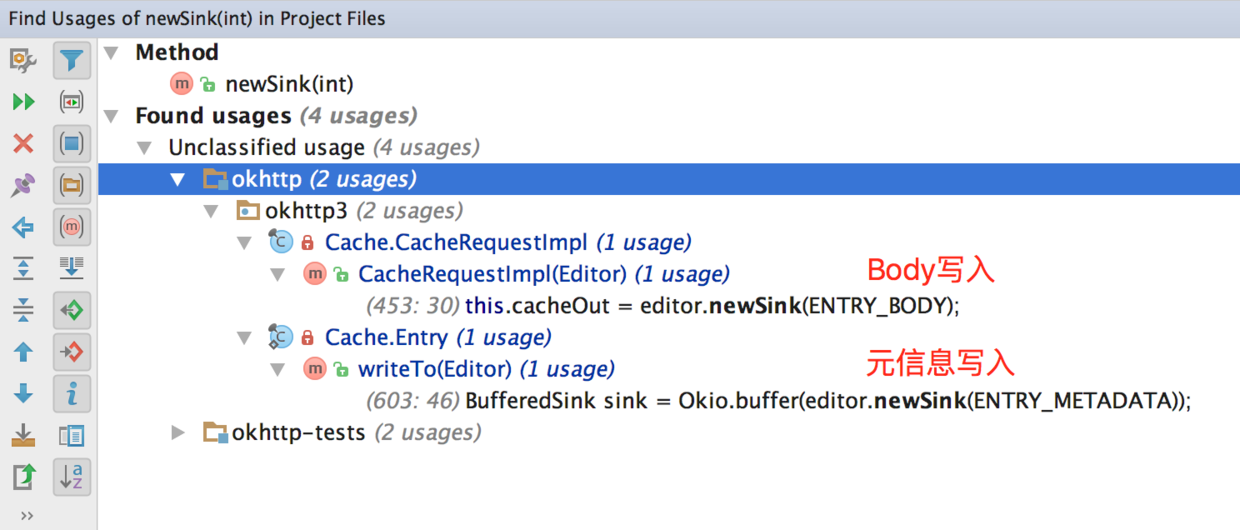
5b2f52377611dc6201a1871bdb997466.1

journal

.....

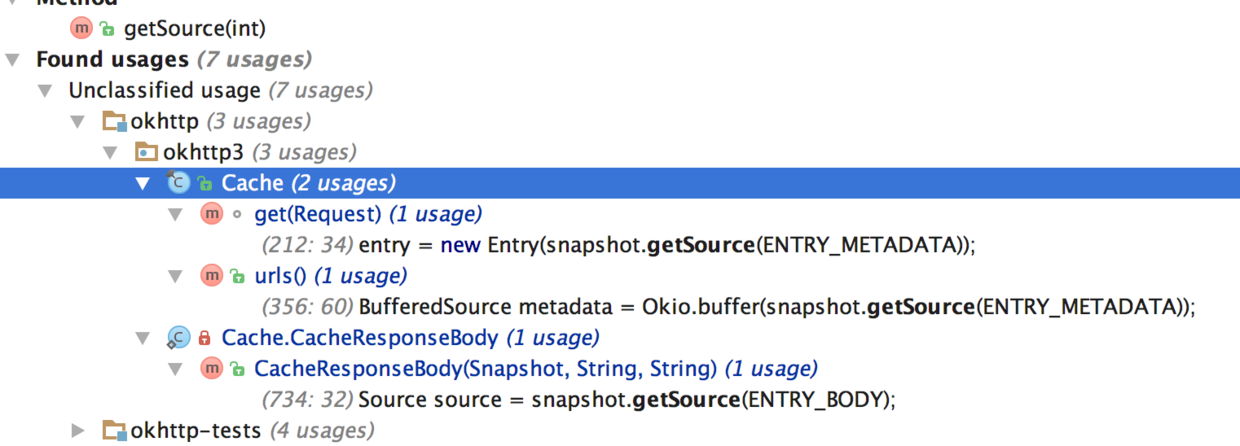
[DiskLruCache.Editor](https://github.com/square/okhttp/blob/d662c1a82851800c46ad8ede2d9d10d10427fdad/okhttp/src/main/java/okhttp3/internal/DiskLruCache.java#L816)对工具类FileSystem进行进一步的封装，它以DiskLruCache.Entry作为构造参数，通过操控Entry中维护的数组，对外暴露source/sink，为上层的java对象与文件的转换提供基于okio的流操作，我们可以通过对它的两个方法进行FindUsage查询获得OkHttp关于文件读写的全部场景

* 写入场景：第一个位置是写入元信息，也就是写入末位是0的文件中，是序列化的过程；第二个位置是写入body，也就是写入末位是1的文件中，是存二进制的过程；



OkHttp Sink to file

* 读取场景：读取时，需要获取快照，通过调用链分析如下



okhttp snapshot source

### 3.3. 序列化与反序列化(Cache.Entry)

文件存储本质上也是序列化与反序列化的过程。本部分提供了下图的转变

Resonse(java对象) <--- Cache.Entry ---> source/sink(文件io)

代码部分不复杂，与上面的findusage位置相同，可以概括下：

如果信息本身就是二进制，就直接写到文件中；如果是文本信息，按照预设的格式写入即可。

至于序列化后的东西到底是什么，可以直接在shell下运行cat命令或者打开文本编辑器进行输出查看。

注意这里的Cache.Entry与上面的DiskLruCache.Entry是两个完全不同的对象

### 3.4 缓存的自动清理

在DiskLruCache初始化时，将[建立](https://github.com/square/okhttp/blob/d662c1a82851800c46ad8ede2d9d10d10427fdad/okhttp/src/main/java/okhttp3/internal/DiskLruCache.java#L252)线程池，最少零个线程，最大一个线程，线程空闲可以活60s，线程名叫做"OkHttp DiskLruCache"，当JVM退出时，线程自动结束。

new ThreadPoolExecutor(0, 1, 60L, TimeUnit.SECONDS,

new LinkedBlockingQueue<Runnable>(), Util.threadFactory("OkHttp DiskLruCache", true))

当需要清理时，执行清理任务，它将在每次get/set后调用

private final Runnable cleanupRunnable = new Runnable() {

public void run() {

synchronized (DiskLruCache.this) {

if (!initialized | closed) {

return; // Nothing to do

}

try {

//遍历LRU缓存(从旧到新进行遍历map),并删除文件

//直到小于MaxSize为止

trimToSize();

if (journalRebuildRequired()) {

rebuildJournal();

redundantOpCount = 0;

}

} catch (IOException e) {

throw new RuntimeException(e);

}

}

}

};

## 总结

1. OkHttp通过对文件进行了多次封装，实现了非常简单的I/O操作
2. OkHttp通过对请求url进行md5实现了与文件的映射，实现写入，删除等操作
3. OkHttp内部维护着清理线程池，实现对缓存文件的自动清理

#### OkHttp3源码分析[任务队列]

本文目录：

1. 线程池基础
2. 反向代理模型
3. OkHttp的任务调度

看过Wiki的都知道OkHttp拥有2种运行方式，一种是同步阻塞调用并直接返回的形式，另一种是通过内部线程池分发调度实现非阻塞的异步回调。本文主要分析第二种，即OkHttp在多并发网络下的分发调度过程。本文主要分析的是Dispatcher对象

## 1. 线程池基础

在初学Java的时候，各位可能会用new Thread + Handler来写异步任务，它的坑网上已经烂大街了，比如不能自动关闭，迷之缩进难以维护，导致目前开发者几乎不怎么用它。而现在很多框架，比如Picasso，Rxjava等，都帮我们写好了对应场景的线程池，但是线程池到底有什么好呢？

### 1.1. 线程池好处都有啥

线程池的关键在于线程复用以减少非核心任务的损耗。下面内容是引用[IBM知识库](https://www.ibm.com/developerworks/cn/java/l-threadPool/)中的例子：

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。但如果对多线程应用不当，会增加对单个任务的处理时间。可以举一个简单的例子：  
假设在一台服务器完成一项任务的时间为T

T1 创建线程的时间

T2 在线程中执行任务的时间，包括线程间同步所需时间

T3 线程销毁的时间

显然T ＝ T1＋T2＋T3。注意这是一个极度简化的假设。  
可以看出T1,T3是多线程本身的带来的开销（在Java中，通过映射pThead，并进一步通过SystemCall实现native线程），我们渴望减少T1,T3所用的时间，从而减少T的时间。但一些线程的使用者并没有注意到这一点，所以在程序中频繁的创建或销毁线程，这导致T1和T3在T中占有相当比例。显然这是突出了线程的弱点（T1，T3），而不是优点（并发性）。

池技术正是关注如何缩短或调整T1，T3时间的技术，从而提高服务器程序性能的。

1. 通过对线程进行缓存，减少了创建销毁的时间损失
2. 通过控制线程数量阀值，减少了当线程过少时带来的CPU闲置（比如说长时间卡在I/O上了）与线程过多时对JVM的内存与线程切换时系统调用的压力

类似的还有Socket连接池、[DB连接池](https://github.com/alibaba/druid)、CommonPool(比如Jedis)等技术。

在Java中，我们可以通过线程池工厂或者自定义参数来创建线程池，这些[教程](http://www.jianshu.com/search?q=%E7%BA%BF%E7%A8%8B%E6%B1%A0&page=1&type=notes)就不讲了

### 1.2. OkHttp配置的线程池

在OkHttp，使用如下构造了单例线程池

public synchronized ExecutorService executorService() {

if (executorService == null) {

executorService = new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>(), Util.threadFactory("OkHttp Dispatcher", false));

}

return executorService;

}

参数说明如下：

* int corePoolSize: 最小并发线程数，这里并发同时包括空闲与活动的线程，如果是0的话，空闲一段时间后所有线程将全部被销毁。
* int maximumPoolSize: 最大线程数，当任务进来时可以扩充的线程最大值，当大于了这个值就会根据丢弃处理机制来处理
* long keepAliveTime: 当线程数大于corePoolSize时，多余的空闲线程的最大存活时间，类似于HTTP中的Keep-alive
* TimeUnit unit: 时间单位，一般用秒
* BlockingQueue<Runnable> workQueue: 工作队列，先进先出，可以看出并不像Picasso那样设置优先队列。
* ThreadFactory threadFactory: 单个线程的工厂，可以打Log，设置Daemon(即当JVM退出时，线程自动结束)等

可以看出，在Okhttp中，构建了一个阀值为[0, Integer.MAX\_VALUE]的线程池，它不保留任何最小线程数，随时创建更多的线程数，当线程空闲时只能活60秒，它使用了一个不存储元素的阻塞工作队列，一个叫做"OkHttp Dispatcher"的线程工厂。

也就是说，在实际运行中，当收到10个并发请求时，线程池会创建十个线程，当工作完成后，线程池会在60s后相继关闭所有线程。

在RxJava的Schedulers.io()中，也有类似的设计，只不过是线程池的池，最小的线程数量控制，不设上限的最大线程，以保证I/O任务中高阻塞低占用的过程中，不会长时间卡在阻塞上，有兴趣的可以分析RxJava中4种不同场景的Schedulers

## 反向代理模型

在OkHttp中，使用了与Nginx类似的反向代理与分发技术，这是典型的单生产者多消费者问题。

我们知道在Nginx/SLB中，用户通过HTTP(Socket)访问前置的服务器，服务器会添加Header并自动转发请求给后端集群，接着返回数据结果给用户(比如简书上次挂了也显示了Nginx报错)。通过将工作分配给多个后台(无状态的)服务器并共享Session，可以提高服务的负载均衡能力，实现非阻塞、高可用、高并发连接，避免资源全部放到一台服务器而带来的负载，速度，在线率等影响。

Nginx Load balancing

而在OkHttp中，非常类似于上述场景，它使用[Dispatcher](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L38-L38)作为任务的派发器，线程池对应多台后置服务器，用AsyncCall对应Socket请求，用Deque<readyAsyncCalls>对应Nginx的内部缓存

Okhttp Dispatcher

具体成员如下

* maxRequests = 64: 最大并发请求数为64
* maxRequestsPerHost = 5: 每个主机最大请求数为5
* Dispatcher: 分发者，也就是生产者（默认在主线程）
* AsyncCall: 队列中需要处理的Runnable（包装了异步回调接口）
* ExecutorService：消费者池（也就是线程池）
* Deque<readyAsyncCalls>：缓存（用数组实现，可自动扩容，无大小限制）
* Deque<runningAsyncCalls>：正在运行的任务，仅仅是用来引用正在运行的任务以判断并发量，注意它并不是消费者缓存

通过将请求任务分发给多个线程，可以显著的减少I/O等待时间

## OkHttp的任务调度

当我们希望使用OkHttp的异步请求时，一般进行如下构造

OkHttpClient client = new OkHttpClient.Builder().build();

Request request = new Request.Builder()

.url("http://qq.com").get().build();

client.newCall(request).enqueue(new Callback() {

@Override public void onFailure(Call call, IOException e) {

}

@Override public void onResponse(Call call, Response response) throws IOException {

}

});

当HttpClient的请求[入队](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L110-L110)时，根据代码，我们可以发现实际上是Dispatcher进行了[入队](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L109)操作

synchronized void enqueue(AsyncCall call) {

if (runningAsyncCalls.size() < maxRequests && runningCallsForHost(call) < maxRequestsPerHost) {

//添加正在运行的请求

runningAsyncCalls.add(call);

//线程池执行请求

executorService().execute(call);

} else {

//添加到缓存队列排队等待

readyAsyncCalls.add(call);

}

}

可以发现请求是否进入缓存的条件如下：

(runningRequests<64 && runningRequestsPerHost<5)

如果满足条件，那么就直接把AsyncCall直接加到runningCalls的队列中，并在线程池中执行（线程池会根据当前负载自动创建，销毁，缓存相应的线程）。反之就放入readyAsyncCalls进行缓存等待。

我们再分析请求元素AsyncCall（它实现了Runnable接口），它[内部实现](https://github.com/square/okhttp/blob/8ff37250310e8d2f9e73293199b3b6e42ec45b0f/okhttp/src/main/java/okhttp3/RealCall.java#L124)的execute方法如下

@Override protected void execute() {

boolean signalledCallback = false;

try {

//执行耗时IO任务

Response response = getResponseWithInterceptorChain(forWebSocket);

if (canceled) {

signalledCallback = true;

//回调，注意这里回调是在线程池中，而不是想当然的主线程回调

responseCallback.onFailure(RealCall.this, new IOException("Canceled"));

} else {

signalledCallback = true;

//回调，同上

responseCallback.onResponse(RealCall.this, response);

}

} catch (IOException e) {

if (signalledCallback) {

// Do not signal the callback twice!

logger.log(Level.INFO, "Callback failure for " + toLoggableString(), e);

} else {

responseCallback.onFailure(RealCall.this, e);

}

} finally {

//最关键的代码

client.dispatcher().finished(this);

}

}

当任务执行完成后，无论是否有异常，finally代码段总会被执行，也就是会调用Dispatcher的[finished](https://github.com/square/okhttp/blob/7826bcb2fb1facb697a4c512776756c05d8c9deb/okhttp/src/main/java/okhttp3/Dispatcher.java#L137)函数，打开源码，发现它将正在运行的任务Call从队列runningAsyncCalls中移除后，接着执行promoteCalls()函数

private void promoteCalls() {

//如果目前是最大负荷运转，接着等

if (runningAsyncCalls.size() >= maxRequests) return; // Already running max capacity.

//如果缓存等待区是空的，接着等

if (readyAsyncCalls.isEmpty()) return; // No ready calls to promote.

for (Iterator<AsyncCall> i = readyAsyncCalls.iterator(); i.hasNext(); ) {

AsyncCall call = i.next();

if (runningCallsForHost(call) < maxRequestsPerHost) {

//将缓存等待区最后一个移动到运行区中，并执行

i.remove();

runningAsyncCalls.add(call);

executorService().execute(call);

}

if (runningAsyncCalls.size() >= maxRequests) return; // Reached max capacity.

}

}

这样，就主动的把缓存队列向前走了一步，而没有使用互斥锁等复杂编码

## Summary

通过上述的分析，我们知道了：

1. OkHttp采用Dispatcher技术，类似于Nginx，与线程池配合实现了高并发，低阻塞的运行
2. Okhttp采用Deque作为缓存，按照入队的顺序先进先出
3. OkHttp最出彩的地方就是在try/finally中调用了finished函数，可以主动控制等待队列的移动，而不是采用锁或者wait/notify，极大减少了编码复杂性

### 8.java多线程的实现方式

[**JAVA多线程实现的四种方式**](http://www.cnblogs.com/felixzh/p/6036074.html)

[Java](http://lib.csdn.net/base/javaee)多线程实现方式主要有四种：继承Thread类、实现Runnable接口、实现Callable接口通过FutureTask包装器来创建Thread线程、使用ExecutorService、Callable、Future实现有返回结果的多线程。

其中前两种方式线程执行完后都没有返回值，后两种是带返回值的。

**1、继承Thread类创建线程**

Thread类本质上是实现了Runnable接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过Thread类的start()实例方法。start()方法是一个native方法，它将启动一个新线程，并执行run()方法。这种方式实现多线程很简单，通过自己的类直接extend Thread，并复写run()方法，就可以启动新线程并执行自己定义的run()方法。例如：

[](javascript:void(0);)

public class MyThread extends Thread {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

MyThread myThread1 = new MyThread();

MyThread myThread2 = new MyThread();

myThread1.start();

myThread2.start();

[](javascript:void(0);)

**2、实现Runnable接口创建线程**

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，可以实现一个Runnable接口，如下：

为了启动MyThread，需要首先实例化一个Thread，并传入自己的MyThread实例：

事实上，当传入一个Runnable target参数给Thread后，Thread的run()方法就会调用target.run()，参考JDK源代码：

**3、实现Callable接口通过FutureTask包装器来创建Thread线程**

Callable接口（也只有一个方法）定义如下：

public interface Callable<V> {

V call（） throws Exception; }

[](javascript:void(0);)

public class SomeCallable<V> extends OtherClass implements Callable<V> {

@Override

public V call() throws Exception {

// TODO Auto-generated method stub

return null;

}

}

[](javascript:void(0);)

[](javascript:void(0);)

Callable<V> oneCallable = new SomeCallable<V>();

//由Callable<Integer>创建一个FutureTask<Integer>对象：

FutureTask<V> oneTask = new FutureTask<V>(oneCallable);

//注释：FutureTask<Integer>是一个包装器，它通过接受Callable<Integer>来创建，它同时实现了Future和Runnable接口。

//由FutureTask<Integer>创建一个Thread对象：

Thread oneThread = new Thread(oneTask);

oneThread.start();

//至此，一个线程就创建完成了。

[](javascript:void(0);)

**4、使用ExecutorService、Callable、Future实现有返回结果的线程**

ExecutorService、Callable、Future三个接口实际上都是属于Executor框架。返回结果的线程是在JDK1.5中引入的新特征，有了这种特征就不需要再为了得到返回值而大费周折了。而且自己实现了也可能漏洞百出。

可返回值的任务必须实现Callable接口。类似的，无返回值的任务必须实现Runnable接口。

执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。

注意：get方法是阻塞的，即：线程无返回结果，get方法会一直等待。

再结合线程池接口ExecutorService就可以实现传说中有返回结果的多线程了。

下面提供了一个完整的有返回结果的多线程测试例子，在JDK1.5下验证过没问题可以直接使用。代码如下：

[](javascript:void(0);)

import java.util.concurrent.\*;

import java.util.Date;

import java.util.List;

import java.util.ArrayList;

/\*\*

\* 有返回值的线程

\*/

@SuppressWarnings("unchecked")

public class Test {

public static void main(String[] args) throws ExecutionException,

InterruptedException {

System.out.println("----程序开始运行----");

Date date1 = new Date();

int taskSize = 5;

// 创建一个线程池

ExecutorService pool = Executors.newFixedThreadPool(taskSize);

// 创建多个有返回值的任务

List<Future> list = new ArrayList<Future>();

for (int i = 0; i < taskSize; i++) {

Callable c = new MyCallable(i + " ");

// 执行任务并获取Future对象

Future f = pool.submit(c);

// System.out.println(">>>" + f.get().toString());

list.add(f);

}

// 关闭线程池

pool.shutdown();

// 获取所有并发任务的运行结果

for (Future f : list) {

// 从Future对象上获取任务的返回值，并输出到控制台

System.out.println(">>>" + f.get().toString());

}

Date date2 = new Date();

System.out.println("----程序结束运行----，程序运行时间【"

+ (date2.getTime() - date1.getTime()) + "毫秒】");

}

}

class MyCallable implements Callable<Object> {

private String taskNum;

MyCallable(String taskNum) {

this.taskNum = taskNum;

}

public Object call() throws Exception {

System.out.println(">>>" + taskNum + "任务启动");

Date dateTmp1 = new Date();

Thread.sleep(1000);

Date dateTmp2 = new Date();

long time = dateTmp2.getTime() - dateTmp1.getTime();

System.out.println(">>>" + taskNum + "任务终止");

return taskNum + "任务返回运行结果,当前任务时间【" + time + "毫秒】";

}

}

[](javascript:void(0);)

代码说明：

上述代码中Executors类，提供了一系列工厂方法用于创建线程池，返回的线程池都实现了ExecutorService接口。

public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

ExecutoreService提供了submit()方法，传递一个Callable，或Runnable，返回Future。如果Executor后台线程池还没有完成Callable的计算，这调用返回Future对象的get()方法，会阻塞直到计算完成。

### 9.android中的lru算法实现

为LinkedHashMap是继承自HashMap，因此LinkedHashMap：

（1）同样是基于散列表实现。

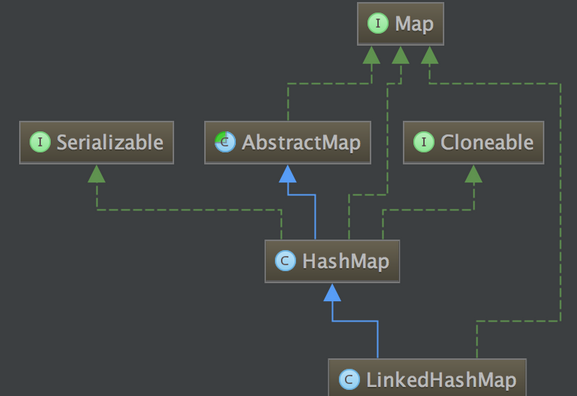
（2）同时实现了Serializable 和 Cloneable接口，支持序列化和克隆。

（3）并且同样不是线程安全的。

区别是其内部维护了一个双向循环链表，该链表是有序的，可以按元素插入顺序或元素最近访问顺序(LRU)排列。

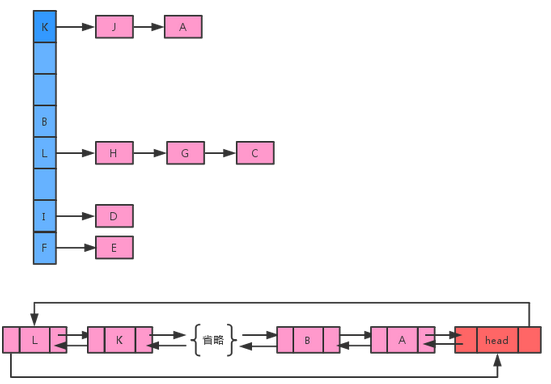
我们在[常见的内存泄漏以及解决方案（二）](http://blog.csdn.net/seu_calvin/article/details/52351062)中介绍的LruCache类就是基于LinkedHashMap实现的。

LinkedHashMap 类层次结构如下所示：



**1．  LinkedHashMap数据存储格式**

下面这张图来自于[BridgeGeorge](http://my.csdn.net/ylyg050518)的博客，省的自己画了。



如上图所示，假设LinkedHashMap进行put操作分别将ABCDEFGHIGKL，共12个KV。LinkedHashMap不仅像HashMap那样对其进行基于哈希表和单链表的Entry数组+ next链表的存储方式，而且还结合了LinkedList的优点，为每个Entry节点增加了前驱和后继，并增加了一个为null头结点，构造了一个双向循环链表。

也就是说，每次put进来KV，除了将其保存到对哈希表中的对应位置外，还要将其插入到双向循环链表的尾部。

**2.  LinkedHashMap的构造方法**

**[java]** view plain copy

1. **public** LinkedHashMap(**int** initialCapacity, **float** loadFactor) {
2. **super**(initialCapacity, loadFactor);
3. accessOrder = **false**;
4. }

若未指定初始容量initialCapacity，则默认为使用HashMap的初始容量，即16。若未指定加载因子loadFactor，则默认为0.75。

accessOrder默认为faslse。这里需要介绍一下这个布尔值，它是双向链表中元素排序规则的标志位。

（1）accessOrder若为false，遍历双向链表时，是按照插入顺序排序。

（2）accessOrder若为true，表示双向链表中的元素按照访问的先后顺序排列，最先遍历到（链表头）的是最近最少使用的元素。

后面会详细讲解这个标志位的作用原理。

**3．  LinkedHashMap的put操作**

**3.1  Key已存在的情况**

在HashMap的put方法中，在发现插入的key已经存在时，除了做替换工作，还会调用recordAccess()方法，在HashMap中该方法为空。LinkedHashMap覆写了该方法，（调用LinkedHashmap覆写的get方法时，也会调用到该方法），LinkedHashmap并没有覆写HashMap中的put方法，recordAccess()在LinkedHashMap中的实现如下：

**[java]** view plain copy

1. **void** recordAccess(HashMap<K,V> m) {
2. LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
3. //判断accessOrder是否为true
4. //将当前访问的Entry放置到双向循环链表的尾部，以标明最近访问
5. **if** (lm.accessOrder) {
6. lm.modCount++;
7. remove();
8. addBefore(lm.header);
9. }
10. }
11. //双向循环链表中，将当前的Entry插入到existing Entry的前面
12. **private** **void** addBefore(Entry<K,V> existingEntry) {
13. after  = existingEntry;
14. before = existingEntry.before;
15. before.after = **this**;
16. after.before = **this**;
17. }

**3.2  Key不存在的情况**

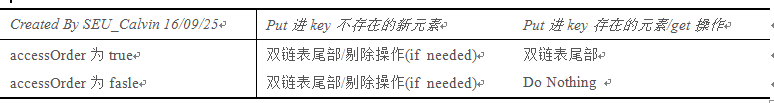
在put新Entry的过程中，如果发现key不存在时，除了将新Entry放到哈希表的相应位置，还会调用addEntry方法，它会调用creatEntry方法，该方法将新插入的元素放到双向链表的尾部，这样做既符合插入的先后顺序，又符合了访问的先后顺序。

**[java]** view plain copy

1. //覆写HashMap中的addEntry方法
2. //在插入的key不存在的情况下，要调用addEntry插入新的Entry
3. **void** addEntry(**int** hash, K key, V value, **int** bucketIndex) {
4. **super**.addEntry(hash, key, value, bucketIndex);
5. //如果有必要，则删除掉该近期最少使用的节点，
6. //这要看对removeEldestEntry的覆写，由于默认为false，因此默认是不做任何处理
7. Entry<K,V> eldest = header.after;
8. **if** (removeEldestEntry(eldest)) {
9. removeEntryForKey(eldest.key);
10. }
11. }
13. **protected** **boolean** removeEldestEntry(Map.Entry<K,V> eldest) {
14. **return** **false**;
15. }
17. **void** createEntry(**int** hash, K key, V value, **int** bucketIndex) {
18. //创建新的Entry，和HashMap一样将其插入到哈希表的相应位置
19. HashMap.Entry<K,V> old = table[bucketIndex];
20. Entry<K,V> e = **new** Entry<>(hash, key, value, old);
21. table[bucketIndex] = e;
22. //并将其移到双向链表的尾部
23. e.addBefore(header);
24. size++;
25. }

在上面的addEntry方法中有一个removeEldestEntry方法，这个方法可以被覆写，比如可以将该方法覆写为如果设定的内存已满，则返回true，这样就可以将最近最少使用的节点（header后的节点）删除掉。

这里为了方便对比总结，我把accessOrder标志位的作用原理做了个表，描述了一些操作对双链表中[**数据结构**](http://lib.csdn.net/base/datastructure)的影响，哈希表中元素该怎么处理还怎么处理，和HashMap是一致的。



从总结的上表来看，只要是put进来的新元素，不管accessOrder标志位是什么，均将新元素放到双链表尾部，并且可以在需要实现Lru[**算法**](http://lib.csdn.net/base/datastructure)时时覆写removeEldestEntry方法，剔除最近最少使用的节点。

还有两种情况，get获取元素、还有put进Key已经存在的元素，即调用recordAccess的这两种情况下，这个时候标志位就起作用了，accessOrder为fasle时，什么也不做，也就是说当我们放入已经存在Key的键值对或get操作时，它在双链表中的位置是不会变的。accessOrder设置为true时，上述两种情况会将相关元素放置到双链表的尾部。在缓存的角度来看，这就是所谓的“脏数据”，即最近被访问过的数据，因此在需要清理内存时（添加进新元素时），就可以将双链表头节点（空节点）后面那个节点剔除。

**4．  LinkedHashMap的get操作**

**[java]** view plain copy

1. //覆写HashMap中的get方法，通过getEntry方法获取Entry对象
2. **public** V get(Object key) {
3. Entry<K,V> e = (Entry<K,V>)getEntry(key);
4. **if** (e == **null**)
5. **return** **null**;
6. e.recordAccess(**this**);
7. **return** e.value;
8. }

通过前面的分析，果然在get中，除了正常的get逻辑，还调用了recordAccess()方法，这个方法的逻辑我们刚刚分析过了，和put进的元素key冲突的情况是一样的，这里就不赘述了。

**5．  LinkedHashMap的清空操作**

**[java]** view plain copy

1. //清空HashMap的同时，将双向链表还原为只有头结点的空链表
2. **public** **void** clear() {
3. **super**.clear();
4. header.before = header.after = header;
5. }

**6．  HashMap和LinkedHashMap的关系和比较**

（1）LinkedHashMap继承自HashMap，HashMap的属性它都有，什么线程不安全，支持null等等。

（2）LinkedHashMap比HashMap多维护了一个双向循环链表。很明显，如果前面写的你看懂了，那么LinkedHashMap中维护了数据的两种排序方式，一个是基于数据插入顺序（默认的方式，将新加入的元素置于双链表的尾部），一种是基于Lru算法（将加入的不管新旧的数据，或者get()到的数据都放在双链表尾部，以标识为脏数据）。

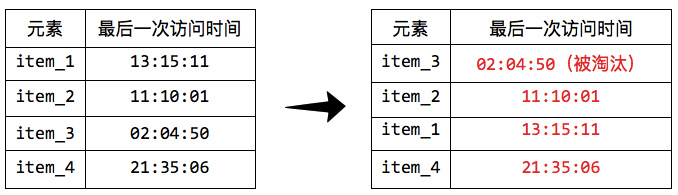
**一、简述**

LRU（Least Recently Used），注意L代表的不是Latest，翻译成中文通常叫：近期最少使用[**算法**](http://lib.csdn.net/base/datastructure)、最近最少使用算法。LRU与LFU（Least Frequently Used）、FIFO（First In First Out）是3种常用的[缓存算法](https://en.wikipedia.org/wiki/Cache_algorithms#LRU)（页面置换算法）。缓存算法的应用场景有很多，例如[**操作系统**](http://lib.csdn.net/base/operatingsystem)在物理内存不足时触发的磁盘交换、CPU中[L1、L2、L3缓存](http://baike.baidu.com/link?url=BmxMH3Wd-zLZxID6jgCU5ejo3fgKrpCR1VseIQZGB-qS7-oYAxNlbDlYS8NFrZEnIxhId1jtQCrdlIIrGd4QqK)淘汰替换、超市中畅销货品在货架的摆放位置优化等。缓存算法的实现和变种也有很多，这里只针对LRU算法抛砖引玉。

**二、算法原理**

缓存算法的根本目的，是淘汰掉“最不常用”的元素。LRU算法淘汰的是截止当前缓存区中最久没有被访问过的元素，这意味着“最后一次访问时间”是LRU算法决定是否淘汰元素的唯一标准。

因此，假设我们记录下每个元素最后一次被访问的时间戳，并按照此时间戳早晚进行排序，那么，时间最早的元素应该被淘汰掉（如图1）。所以，将LRU准确地翻译成中文，应该是：距今最久未用淘汰算法。

图1：最后一次访问时间最早的元素首先被淘汰

这里补充一点，LRU算法并不是绝对公平的。举个例子，假如某个元素仅仅在淘汰执行前被访问了1次，在之前的数百万次的请求中从未被访问，按照LRU的算法逻辑，它仍然会留存下来，这其实无法准确描述“是否常用”这一特性。“是否常用”其实在不同的业务场景有不同的定义，大家可以细细体会。

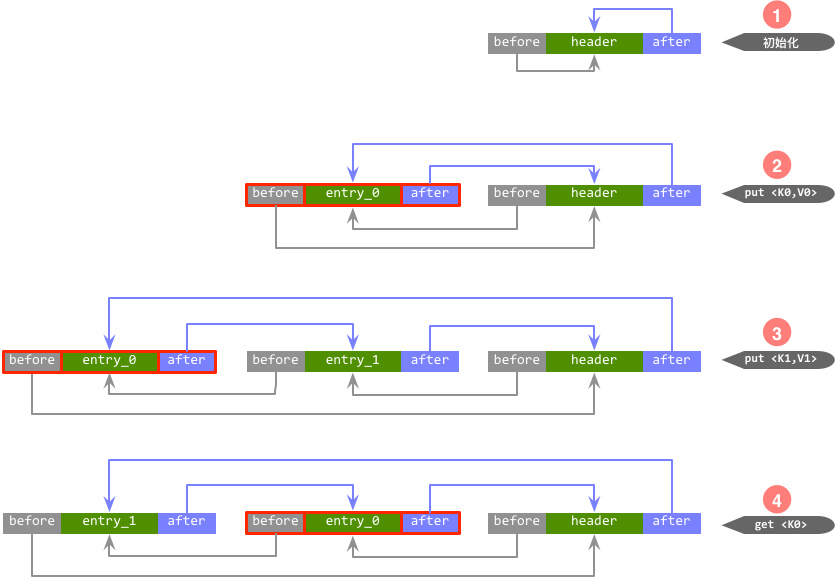
**三、LRU的实现机制**

依照LRU算法原理，其实现机制并不复杂，仅仅需要根据元素的最后一次访问时间排序即可。但当运用在1个生产环境的缓存系统中时，会面临以下几个工程问题：

1. 在缓存访问频次极高的情况下，时间戳即使精确到纳秒，依然存在大量的相同时间戳，排序无效。若采用递增ID替代时间戳，则存在溢出的风险。
2. 每次进行缓存淘汰时，都需要将缓存区所有元素进行排序。当元素数极多时，缓存系统的性能将急剧下降，CPU耗费极高。
3. 在元素本身占用内存不大的情况下，附加的“时间戳/自增ID”甚至在内存占用上喧宾夺主。

最佳的实现思路是利用链表的有序特性，将缓存元素的“按最后一次访问时间戳排序”巧妙地转换为其在链表中的相对顺序。因为LRU算法关心的并不是元素的绝对访问时间，而是元素被访问的先后顺序。即：元素被命中时，移到链表头。执行淘汰时，从链表尾开始淘汰。因此，LRU算法实现的核心[**数据结构**](http://lib.csdn.net/base/datastructure)是：循环链表。

但对于一个<K,V>缓存系统而言，仅有链表结构是不够的。链表虽然解决了缓存淘汰问题，但缓存访问却需要每次都从链表头开始遍历。JDK中的[**Java**](http://lib.csdn.net/base/java).util.LinkedHashMap给出了最佳的实现（链表也是Linked这一前缀的由来），同时解决了元素的快速访问与缓存淘汰问题。



### 10.单例的双锁机制

为什么要在多线程中创建单例模式的时候要进行双重锁定？先回顾一下双重锁定的代码块。

**[java]** [view plain](http://blog.csdn.net/nyist327/article/details/49301401) [copy](http://blog.csdn.net/nyist327/article/details/49301401)

1. **public** **class** SingleTon {
3. **private** **static** SingleTon singleTon = **null**;

6. **public** SingleTon() {
7. // TODO Auto-generated constructor stub
8. }

11. **public** **static** SingleTon getInstance(){
12. **if** (singleTon == **null**) {
13. **synchronized** (SingleTon.**class**) {
14. **if** (singleTon == **null**) {
15. singleTon = **new** SingleTon();
16. }
17. }
18. }
19. **return** singleTon;
20. }
22. }

**为何要使用双重检查锁定呢？上文已经大概说了一下。**

考虑这样一种情况，就是有两个线程同时到达，即同时调用 getInstance() 方法，

此时由于 singleTon == null ，所以很明显，两个线程都可以通过第一重的 singleTon == null ，

进入第一重 if 语句后，由于存在锁机制，所以会有一个线程进入 lock 语句并进入第二重 singleTon == null ，

而另外的一个线程则会在 lock 语句的外面等待。

而当第一个线程执行完 new  SingleTon（）语句后，便会退出锁定区域，此时，第二个线程便可以进入 lock 语句块，

此时，如果没有第二重 singleTon == null 的话，那么第二个线程还是可以调用 new  SingleTon （）语句，

这样第二个线程也会创建一个 SingleTon实例，这样也还是违背了单例模式的初衷的，

所以这里必须要使用双重检查锁定。

细心的朋友一定会发现，如果我去掉第一重 singleton == null ，程序还是可以在多线程下完好的运行的，

考虑在没有第一重 singleton == null 的情况下，

当有两个线程同时到达，此时，由于 lock 机制的存在，第一个线程会进入 lock 语句块，并且可以顺利执行 new SingleTon（），

当第一个线程退出 lock 语句块时， singleTon 这个静态变量已不为 null 了，所以当第二个线程进入 lock 时，

还是会被第二重 singleton == null 挡在外面，而无法执行 new Singleton（），

**所以在没有第一重 singleton == null 的情况下，也是可以实现单例模式的？那么为什么需要第一重 singleton == null 呢？**

这里就涉及一个性能问题了，因为对于单例模式的话，new SingleTon（）只需要执行一次就 OK 了，

而如果没有第一重 singleTon == null 的话，每一次有线程进入 getInstance（）时，均会执行锁定操作来实现线程同步，

这是非常耗费性能的，而如果我加上第一重 singleTon == null 的话，

那么就只有在第一次，也就是 singleTton ==null 成立时的情况下执行一次锁定以实现线程同步，

而以后的话，便只要直接返回 Singleton 实例就 OK 了而根本无需再进入 lock 语句块了，这样就可以解决由线程同步带来的性能问题了。

### 11.Fresco源码解析

**关于 Fresco**

Fresco 是一个强大的图片加载组件。

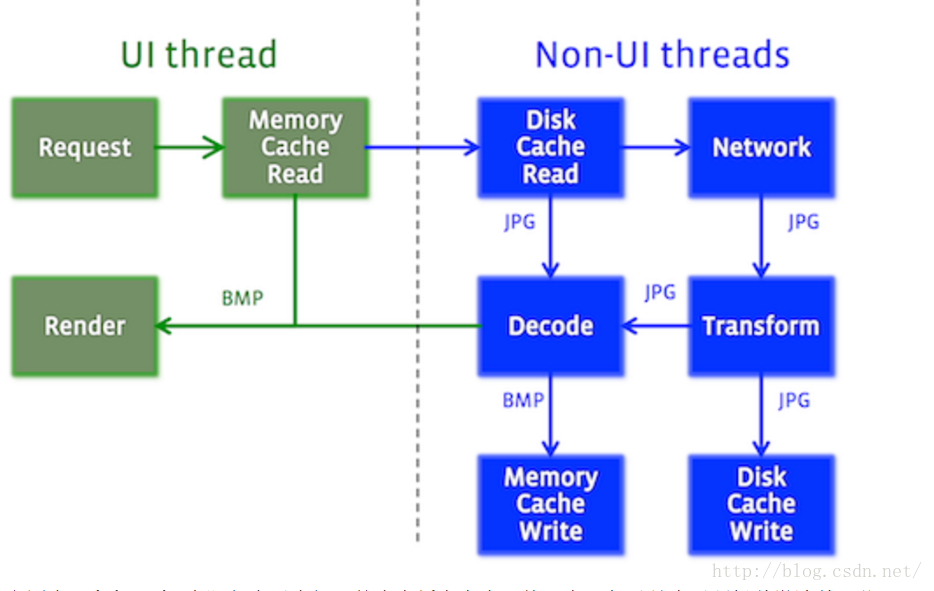
Fresco 中设计有一个叫做 image pipeline 的模块。它负责从网络，从本地文件系统，本地资源加载图片。为了最大限度节省空间和CPU时间，它含有3级缓存设计（2级内存，1级文件）。

Fresco 中设计有一个叫做 Drawees 模块，方便地显示loading图，当图片不再显示在屏幕上时，及时地释放内存和空间占用。

Fresco 支持 Android2.3(API level 9) 及其以上系统。

这么强大而优秀的开源项目，学习学习人家的源码是非常不错的，看了一两天，记录一下自己的笔记与见解。源码比较复杂，我的看法不一定对，如有错误之处欢迎指出，欢迎交流学习。

fresco作为一个图片加载框架肯定离不开数据或者，数据缓存等基本流程。如下图描述那样。



实际上，通过查看源码，Fresco使用的时候是这样的：提交请求------》获取数据

提交请求很容易理解，难点就是获取数据的过程，它获取数据的过程基本是上图的逆过程进行的，就是提交了请求-----》判断有没有缓存内存--------》没有获取网络数据

下面通过源码去看看整个流程，为了不被细节干扰，我们主要看主干线就行了，这样可以把握整体的逻辑流程。

fresco的简单使用是这样的：

**[java]** view plain copy

print?

1. Uri uri = Uri.parse("https://raw.githubusercontent.com/facebook/fresco/gh-pages/static/fresco-logo.png");
2. SimpleDraweeView draweeView = (SimpleDraweeView) findViewById(R.id.my\_image\_view);
3. draweeView.setImageURI(uri);

只传递了图片的url，它会自动帮你把所有事情做了，找到这个draweeView.setImageURI(uri);

**[java]** view plain copy

print?

1. /\*\*
2. \* Displays an image given by the uri.
3. \*
4. \* @param uri uri of the image
5. \* @undeprecate
6. \*/
7. @Override
8. **public** **void** setImageURI(Uri uri) {
9. setImageURI(uri, **null**);
10. }
12. /\*\*
13. \* Displays an image given by the uri.
14. \*
15. \* @param uri uri of the image
16. \* @param callerContext caller context
17. \*/
18. **public** **void** setImageURI(Uri uri, @Nullable Object callerContext) {
19. //controller是com.facebook.drawee.backends.pipeline.PipelineDraweeController
20. DraweeController controller = mSimpleDraweeControllerBuilder
21. .setCallerContext(callerContext)
22. .setUri(uri)
23. .setOldController(getController())
24. .build();
25. setController(controller);
26. }

这里我们必须找到DraweeController controller是哪个，我们看看mSimpleDraweeControllerBuilder是什么。

**[java]** view plain copy

print?

1. **private** **void** init() {
2. Preconditions.checkNotNull(
3. sDraweeControllerBuilderSupplier,
4. "SimpleDraweeView was not initialized!");
5. //可以看到这个初始化的时候初始化了mSimpleDraweeControllerBuilder，是通过sDraweeControllerBuilderSupplier获取的
6. mSimpleDraweeControllerBuilder = sDraweeControllerBuilderSupplier.get();
7. }

然而在初始化的时候初始化了这个sDraweeControllerBuilderSupplier

**[java]** view plain copy

print?

1. /\*\* Initializes {@link SimpleDraweeView} with supplier of Drawee controller builders. \*/
2. **public** **static** **void** initialize(
3. Supplier<? **extends** SimpleDraweeControllerBuilder> draweeControllerBuilderSupplier) {
4. sDraweeControllerBuilderSupplier = draweeControllerBuilderSupplier;
5. }

而这个方法是在Fresco这个类的的初始化方法了被执行的，就是说fresco在初始化的时候就初始化了这个相关配置。

**[java]** view plain copy

print?

1. /\*\* Initializes Fresco with the default config. \*/
2. **public** **static** **void** initialize(Context context) {
3. ImagePipelineFactory.initialize(context);
4. initializeDrawee(context);
5. }
7. /\*\* Initializes Fresco with the specified config. \*/
8. **public** **static** **void** initialize(Context context, ImagePipelineConfig imagePipelineConfig) {
9. ImagePipelineFactory.initialize(imagePipelineConfig);
10. initializeDrawee(context);
11. }
13. **private** **static** **void** initializeDrawee(Context context) {
14. sDraweeControllerBuilderSupplier = **new** PipelineDraweeControllerBuilderSupplier(context);
15. SimpleDraweeView.initialize(sDraweeControllerBuilderSupplier);
17. }

同时 我们知道了mSimpleDraweeControllerBuilder 就是PipelineDraweeControllerBuilderSupplier的get()，返回的，看看这个get的方法：

**[java]** view plain copy

print?

1. @Override
2. **public** PipelineDraweeControllerBuilder get() {
3. **return** newPipelineDraweeControllerBuilder(
4. mContext,
5. mPipelineDraweeControllerFactory,
6. mImagePipeline,
7. mBoundControllerListeners);
8. }

看看这个builder的build的方法返回的是什么：

**[java]** view plain copy

print?

1. /\*\* Builds the specified controller. \*/
2. @Override
3. **public** AbstractDraweeController build() {
4. validate();
6. // if only a low-res request is specified, treat it as a final request.
7. **if** (mImageRequest == **null** && mMultiImageRequests == **null** && mLowResImageRequest != **null**) {
8. mImageRequest = mLowResImageRequest;
9. mLowResImageRequest = **null**;
10. }
12. **return** buildController();
13. }

**[java]** view plain copy

print?

1. /\*\* Builds a regular controller. \*/
2. **protected** AbstractDraweeController buildController() {
3. AbstractDraweeController controller = obtainController();
4. maybeBuildAndSetRetryManager(controller);
5. maybeAttachListeners(controller);
6. **return** controller;
7. }

**[java]** view plain copy

print?

1. @Override
2. **protected** PipelineDraweeController obtainController() {
3. DraweeController oldController = getOldController();
4. PipelineDraweeController controller;
5. **if** (oldController **instanceof** PipelineDraweeController) {
6. controller = (PipelineDraweeController) oldController;
7. controller.initialize(
8. obtainDataSourceSupplier(),
9. generateUniqueControllerId(),
10. getCallerContext());
11. } **else** {
12. mPipelineDraweeControllerFactory.newController(
13. obtainDataSourceSupplier(),
14. generateUniqueControllerId(),
15. getCallerContext());
16. }
17. **return** controller;
18. }

可以看到它通过工厂模式去生成controller：

**[java]** view plain copy

print?

1. **public** PipelineDraweeController newController(
2. Supplier<DataSource<CloseableReference<CloseableImage>>> dataSourceSupplier,
3. String id,
4. Object callerContext) {
5. **return** **new** PipelineDraweeController</span>(
6. mResources,
7. mDeferredReleaser,
8. mAnimatedDrawableFactory,
9. mUiThreadExecutor,
10. dataSourceSupplier,
11. id,
12. callerContext);
13. }

走了那么远，终于可以回去，回到原点看。

**[java]** view plain copy

print?

1. /\*\*
2. \* Displays an image given by the uri.
3. \*
4. \* @param uri uri of the image
5. \* @param callerContext caller context
6. \*/
7. **public** **void** setImageURI(Uri uri, @Nullable Object callerContext) {
8. //controller是com.facebook.drawee.backends.pipeline.PipelineDraweeController
9. DraweeController controller</span> = mSimpleDraweeControllerBuilder
10. .setCallerContext(callerContext)
11. .setUri(uri)
12. .setOldController(getController())
13. .build();
14. setController(controller);
15. }

现在我们知道了一点：这个SimpleDraweeView 使用的控制器是PipelineDraweeController；

**现在问题是setController(controller);后发生了什么，为什么用了这个方法后所有处理都自动完成了呢。**

我们看看这个setController(controller);方法：

**[java]** view plain copy

print?

1. /\*\* Sets the controller. \*/
2. **public** **void** setController(@Nullable DraweeController draweeController) {
3. mDraweeHolder.setController(draweeController);
4. **super**.setImageDrawable(mDraweeHolder.getTopLevelDrawable());
5. }

这里将控制器传进了holder里面。这里简单说一下，fresco采用的是典型的mvc模式，DraweeHierarchy 负责的是跟显示相关的，DraweeController 负责的是后台相关的，DraweeHolder主要是统筹两者的，其他还有一下事件记录等。官方是这么说的：

A holder class for Drawee controller and hierarchy.

返回到上面看这个setController方法：

**[java]** view plain copy

print?

1. /\*

**[java]** view plain copy

print?

1. \* Sets a **new** controller.
2. \*/
3. **public** **void** setController(@Nullable DraweeController draweeController) {
4. **boolean** wasAttached = mIsControllerAttached;
5. **if** (wasAttached) {
6. detatchController();
7. }
8. .....
9. **if** (wasAttached) {
10. <span style="color: rgb(255, 0, 0);">attachController();</span>
11. }
12. }

**[java]** view plain copy

print?

1. **private** **void** attachController() {
2. **if** (mIsControllerAttached) {
3. **return**;
4. }
5. mEventTracker.recordEvent(Event.ON\_ATTACH\_CONTROLLER);
6. mIsControllerAttached = **true**;
7. **if** (mController != **null** &&
8. mController.getHierarchy() != **null**) {
9. mController.onAttach();
10. }
11. }

看到了么，它调用了controller的onAttach()方法，我们已经知道了这个controller是PipelineDraweeController，我们去看看这个方法：

**[java]** view plain copy

print?

1. @Override
2. **public** **void** onAttach() {
3. **if** (FLog.isLoggable(FLog.VERBOSE)) {
4. FLog.v(
5. TAG,
6. "controller %x %s: onAttach: %s",
7. System.identityHashCode(**this**),
8. mId,
9. mIsRequestSubmitted ? "request already submitted" : "request needs submit");
10. }
11. mEventTracker.recordEvent(Event.ON\_ATTACH\_CONTROLLER);
12. Preconditions.checkNotNull(mSettableDraweeHierarchy);
13. mDeferredReleaser.cancelDeferredRelease(**this**);
14. mIsAttached = **true**;
15. **if** (!mIsRequestSubmitted) {
16. submitRequest();//看到没这里进行了提交请求
17. }
18. }

我们看看这个提交的方法：

**[java]** view plain copy

print?

1. **protected** **void** submitRequest() {
2. mEventTracker.recordEvent(Event.ON\_DATASOURCE\_SUBMIT);
3. getControllerListener().onSubmit(mId, mCallerContext);
4. mSettableDraweeHierarchy.setProgress(0, **true**);
5. mIsRequestSubmitted = **true**;
6. mHasFetchFailed = **false**;
7. mDataSource = getDataSource();//获取数据
8. **if** (FLog.isLoggable(FLog.VERBOSE)) {
9. FLog.v(
10. TAG,
11. "controller %x %s: submitRequest: dataSource: %x",
12. System.identityHashCode(**this**),
13. mId,
14. System.identityHashCode(mDataSource));
15. }
16. **final** String id = mId;
17. **final** **boolean** wasImmediate = mDataSource.hasResult();
18. **final** DataSubscriber<T> dataSubscriber =
19. **new** BaseDataSubscriber<T>() {
20. @Override
21. **public** **void** onNewResultImpl(DataSource<T> dataSource) {
22. // isFinished must be obtained before image, otherwise we might set intermediate result
23. // as final image.
24. **boolean** isFinished = dataSource.isFinished();
25. T image = dataSource.getResult();
26. **if** (image != **null**) {
27. onNewResultInternal(id, dataSource, image, isFinished, wasImmediate);
28. } **else** **if** (isFinished) {
29. onFailureInternal(id, dataSource, **new** NullPointerException(), /\* isFinished \*/ **true**);
30. }
31. }
32. @Override
33. **public** **void** onFailureImpl(DataSource<T> dataSource) {
34. onFailureInternal(id, dataSource, dataSource.getFailureCause(), /\* isFinished \*/ **true**);
35. }
36. };
38. mDataSource.subscribe(dataSubscriber, mUiThreadImmediateExecutor);//这个将获取的数据回调到ui线程</span>
39. }

可以看到，这里大概可以知道了，获取到数据，然后将数据提交到ui线程去，那么我们这里关心的有两个，**1.如何获取到数据。 2.提交数据后怎么处理。**

**我们分开去看比较好，这里将获取后数据处理标志为Q1,先不管这个，我们先重点关心它是怎么获取到数据的。**

然后获取到数据，那么 我们看这个方法mDataSource = getDataSource();//获取数据，这里发现是一个抽象方法，找到其实现：

**[java]** view plain copy

print?

1. @Override
2. **protected** DataSource<CloseableReference<CloseableImage>> getDataSource() {
3. **if** (FLog.isLoggable(FLog.VERBOSE)) {
4. FLog.v(TAG, "controller %x: getDataSource", System.identityHashCode(**this**));
5. }
6. **return** mDataSourceSupplier.get();
7. }

这里出现了一个mDataSourceSupplier，发现这个mDataSourceSupplier只有在两个地方传递进去，一个是构造器，还有一个是initialize方法，而这个方法只有在PipelineDraweeControllerBuilder的obtainController()时调用到：

**[java]** view plain copy

print?

1. @Ove<span style="font-family: Arial, Helvetica, sans-serif;">rride</span>

**[java]** view plain copy

print?

1. **protected** PipelineDraweeController obtainController() {
2. DraweeController oldController = getOldController();
3. PipelineDraweeController controller;
4. **if** (oldController **instanceof** PipelineDraweeController) {
5. controller = (PipelineDraweeController) oldController;
6. controller.initialize(
7. obtainDataSourceSupplier(),
8. generateUniqueControllerId(),
9. getCallerContext());
10. } **else** {
11. controller = mPipelineDraweeControllerFactory.newController(
12. obtainDataSourceSupplier(),
13. generateUniqueControllerId(),
14. getCallerContext());
15. }
16. **return** controller;
17. }

可以看到了，在创建那个控制器的时候同时会初始化那个数据提供者Supplier<DataSource<IMAGE>>，我们看看这个东西是怎么创建的：

**[java]** view plain copy

print?

1. /\*\* Gets the top-level data source supplier to be used by a controller. \*/</span>
2. **protected** Supplier<DataSource<IMAGE>> obtainDataSourceSupplier() {
3. **if** (mDataSourceSupplier != **null**) {
4. **return** mDataSourceSupplier;
5. }
7. Supplier<DataSource<IMAGE>> supplier = **null**;
9. // final image supplier;
10. **if** (mImageRequest != **null**) {
11. supplier = getDataSourceSupplierForRequest(mImageRequest);//根据请求获取数据提供者
12. } **else** **if** (mMultiImageRequests != **null**) {
13. supplier = getFirstAvailableDataSourceSupplier(mMultiImageRequests);
14. }
16. // increasing-quality supplier; highest-quality supplier goes first
17. **if** (supplier != **null** && mLowResImageRequest != **null**) {
18. List<Supplier<DataSource<IMAGE>>> suppliers = Lists.newArrayListWithCapacity(2);
19. suppliers.add(supplier);
20. suppliers.add(getDataSourceSupplierForRequest(mLowResImageRequest));
21. supplier = IncreasingQualityDataSourceSupplier.create(suppliers);
22. }
24. // no image requests; use null data source supplier
25. **if** (supplier == **null**) {
26. supplier = DataSources.getFailedDataSourceSupplier(NO\_REQUEST\_EXCEPTION);
27. }
29. **return** supplier;
30. }

**[java]** view plain copy

print?

1. /\*\* Creates a data source supplier for the given image request. \*/
2. **protected** Supplier<DataSource<IMAGE>> getDataSourceSupplierForRequest(REQUEST imageRequest){
3. **return** getDataSourceSupplierForRequest(imageRequest, /\* bitmapCacheOnly \*/ **false**);
4. }

**[java]** view plain copy

print?

* /\*\* Creates a data source supplier for the given image request. \*/
* **protected** Supplier<DataSource<IMAGE>> getDataSourceSupplierForRequest(**final** REQUEST imageRequest,
* **final** **boolean** bitmapCacheOnly) {
* **final** Object callerContext = getCallerContext();
* **return** **new** Supplier<DataSource<IMAGE>>() {<span style="color: rgb(255, 0, 0);">//创建一个新的数据提供者</span>
* @Override
* **public** DataSource<IMAGE> get() {
* **return** getDataSourceForRequest(imageRequest, callerContext, bitmapCacheOnly);//获取数据封装到数据提供者
* }
* @Override
* **public** String toString() {
* **return** Objects.toStringHelper(**this**).add("request", imageRequest.toString()).toString();
* }
* };
* }

看到没有，非常清晰了，这里创建了一个新的数据提供者，而正在获取数据的是getDataSourceForRequest这个方法，这个是抽象方法，很容易找到其实现：

**[java]** view plain copy

print?

* @Override
* **protected** DataSource<CloseableReference<CloseableImage>> getDataSourceForRequest(
* ImageRequest imageRequest,
* Object callerContext,
* **boolean** bitmapCacheOnly) {
* **if** (bitmapCacheOnly) {
* **return** mImagePipeline.fetchImageFromBitmapCache(imageRequest, callerContext);<span style="color: rgb(255, 0, 0);">//这里获取的是缓存的数据</span>
* } **else** {
* **return** mImagePipeline.fetchDecodedImage(imageRequest, callerContext);//这里获取的解码的imge数据，我们关心的是这个
* }
* }

千呼万唤始出来啊，看到了，是通过这里mImagePipeline.fetchDecodedImage获取到数据的，我们关心的是它是如何获取网络数据并返回回来了，下面的才是重头戏，至少是我认为精妙之处。好，接着看。mImagePipeline的作用是提交请求将数据返回回来，这里的这个的初始化是在构造器里进行，这个只有在一个地方调用到，就是在PipelineDraweeControllerBuilderSupplier的get方法：

**[java]** view plain copy

print?

* @Override
* **public** PipelineDraweeControllerBuilder get() {
* **return** **new** PipelineDraweeControllerBuilder(
* mContext,
* mPipelineDraweeControllerFactory,
* mImagePipeline,
* mBoundControllerListeners);
* }

咦，熟悉吧这个方法，就是SimpleDraweeView初始化的时候进行的。

**[java]** view plain copy

print?

* **private** **void** init() {
* Preconditions.checkNotNull(
* sDraweeControllerBuilderSupplier,
* "SimpleDraweeView was not initialized!");
* //可以看到这个初始化的时候初始化了mSimpleDraweeControllerBuilder，是通过sDraweeControllerBuilderSupplier获取的
* mSimpleDraweeControllerBuilder = sDraweeControllerBuilderSupplier.get();
* }

哈哈，这样一来都打通了。

**我们明白了 mImagePipeline是怎么来后，核心是关注mImagePipeline.fetchDecodedImage(imageRequest, callerContext);这个方法是怎么将数据返回回来的，核心啊！！**

**[java]** view plain copy

print?

* /\*\*
* \* Submits a request for execution and returns a DataSource representing the pending decoded
* \* image(s).
* \*
* \* The returned DataSource must be closed once the client has finished with it.
* \* @param imageRequest the request to submit
* \* @return a DataSource representing the pending decoded image(s)
* \*/
* **public** DataSource<CloseableReference<CloseableImage>> fetchDecodedImage(
* ImageRequest imageRequest,
* Object callerContext) {
* //Returns a sequence that can be used for a request for a decoded image
* Producer<CloseableReference<CloseableImage>> producerSequence =
* mProducerSequenceFactory.getDecodedImageProducerSequence(
* imageRequest);//Q2
* **return** submitFetchRequest(producerSequence, imageRequest, callerContext);
* }

这里有两个方法，一个是返回了一个序列，暂时不明白做什么的，一个submitFetchRequest很容易明白，这个方法是返回数据的，我们将前面一个标志为Q2,等下再看，我们先看看这个submitFetchRequest是如何返回数据的：

**[java]** view plain copy

print?

* <span style="font-family: Arial, Helvetica, sans-serif;"> **private** <T> DataSource<CloseableReference<T>> submitFetchRequest(</span>

**[java]** view plain copy

print?

* Producer<CloseableReference<T>> producerSequence,
* ImageRequest imageRequest,
* Object callerContext) {
* SettableProducerContext settableProducerContext = **new** SettableProducerContext(
* imageRequest,
* generateUniqueFutureId(),
* mRequestListener,
* callerContext,
* /\* isPrefetch \*/ **false**,
* imageRequest.getProgressiveRenderingEnabled() ||
* !UriUtil.isNetworkUri(imageRequest.getSourceUri()),
* Priority.HIGH);
* **return** CloseableProducerToDataSourceAdapter.create(
* producerSequence,
* settableProducerContext,
* mRequestListener);
* }

这里有个SettableProducerContext，看了下注释,大概是提供上下文一样的东西，方便取消请求，先不管。

**[java]** view plain copy

print?

* /\*\*
* \* ProducerContext that allows the client to cancel an image request in-flight.
* \*/
* @ThreadSafe
* **public** **class** SettableProducerContext **implements** ProducerContext

那么核心就是这个方法了：

**[java]** view plain copy

print?

* **return** CloseableProducerToDataSourceAdapter.create(
* producerSequence,
* settableProducerContext,
* mRequestListener);

这里传递进去了一个刚才标志为Q2的序列，一个上下文，一个请求监听，看看这个方法源码：

**[java]** view plain copy

print?

* /\*\*
* \* DataSource<CloseableReference<T>> backed by a Producer<CloseableReference<T>>
* \*
* \* @param <T>
* \*/
* @ThreadSafe
* **public** **class** CloseableProducerToDataSourceAdapter<T>
* **extends** AbstractProducerToDataSourceAdapter<CloseableReference<T>> {
* **public** **static** <T> DataSource<CloseableReference<T>> create(
* Producer<CloseableReference<T>> producer,
* SettableProducerContext settableProducerContext,
* RequestListener listener) {
* **return** **new** CloseableProducerToDataSourceAdapter<T>(
* producer, settableProducerContext, listener);
* }

发现主要核心还是在父类里面：

**[java]** view plain copy

print?

* **protected** AbstractProducerToDataSourceAdapter(
* Producer<T> producer,//这是刚才传递进去的Q2
* SettableProducerContext settableProducerContext,
* RequestListener requestListener) {
* mSettableProducerContext = settableProducerContext;
* mRequestListener = requestListener;
* mRequestListener.onRequestStart(
* settableProducerContext.getImageRequest(),
* mSettableProducerContext.getCallerContext(),
* mSettableProducerContext.getId(),
* mSettableProducerContext.isPrefetch());
* //这个createConsumer()是最终返回数据处理的消费者
* producer.produceResults(createConsumer(), settableProducerContext);
* }

**[java]** view plain copy

print?

* **private** Consumer<T> createConsumer() {
* **return** **new** BaseConsumer<T>() {
* @Override//新的数据回调</span>
* **protected** **void** onNewResultImpl(@Nullable T newResult, **boolean** isLast) {
* AbstractProducerToDataSourceAdapter.**this**.onNewResultImpl(newResult, isLast);
* }
* @Override//数据获取失败回调</span>
* **protected** **void** onFailureImpl(Throwable throwable) {
* AbstractProducerToDataSourceAdapter.**this**.onFailureImpl(throwable);
* }
* @Override//请求取消回调
* **protected** **void** onCancellationImpl() {
* AbstractProducerToDataSourceAdapter.**this**.onCancellationImpl();
* }
* };
* }
* **protected** **void** onNewResultImpl(@Nullable T result, **boolean** isLast) {
* **if** (**super**.setResult(result, isLast)) {
* **if** (isLast) {
* mRequestListener.onRequestSuccess(//是调用
* mSettableProducerContext.getImageRequest(),
* mSettableProducerContext.getId(),
* mSettableProducerContext.isPrefetch());
* }
* }
* }
* **private** **void** onFailureImpl(Throwable throwable) {
* **if** (**super**.setFailure(throwable)) {
* mRequestListener.onRequestFailure(//mRequestListener将结果返回，成功或者失败
* mSettableProducerContext.getImageRequest(),
* mSettableProducerContext.getId(),
* throwable,
* mSettableProducerContext.isPrefetch());
* }
* }

这里我们发现了，Q2的producer的 producer.produceResults的方法传递进去一个Consumer，然后会自动将数据通过Consumer返回回来，接着通过requestlistener回调回去。这里可以看到了，实现上返回数据回来的还是Q2的那个方法，这里还有个疑问，这个requestlistener是如何将数据返回出去的呢？我们先解决一下这个疑问。

毫无疑问，在这个方法里传进了这个requestlistsner

**[java]** view plain copy

print?

* **return** CloseableProducerToDataSourceAdapter.create(
* producerSequence,
* settableProducerContext,
* mRequestListener);

发现这个RequestListsner是ImagePipeline的成员变量，它在构造器就进行初始化：

**[java]** view plain copy

print?

* **public** ImagePipeline(
* ProducerSequenceFactory producerSequenceFactory,
* Set<RequestListener> requestListeners,
* Supplier<Boolean> isPrefetchEnabledSupplier,
* MemoryCache<BitmapMemoryCacheKey, CloseableImage, Void> bitmapMemoryCache,
* MemoryCache<CacheKey, PooledByteBuffer, Void> encodedMemoryCache,
* CacheKeyFactory cacheKeyFactory) {
* mIdCounter = **new** AtomicLong();
* mProducerSequenceFactory = producerSequenceFactory;
* mRequestListener = **new** ForwardingRequestListener(requestListeners);
* mIsPrefetchEnabledSupplier = isPrefetchEnabledSupplier;
* mBitmapMemoryCache = bitmapMemoryCache;
* mEncodedMemoryCache = encodedMemoryCache;
* mCacheKeyFactory = cacheKeyFactory;
* }

**[java]** view plain copy

print?

* **public** **class** ForwardingRequestListener **implements** RequestListener {
* **private** **static** **final** String TAG = "ForwardingRequestListener";
* **private** **final** List<RequestListener> mRequestListeners;
* **public** ForwardingRequestListener(
* Set<RequestListener> requestListeners) {
* mRequestListeners = Lists.newArrayListWithCapacity(requestListeners.size());
* **for** (RequestListener requestListener : requestListeners) {
* mRequestListeners.add(requestListener);
* }
* }

通过源码很容易看出来ForwardingRequestListener 维持了一个请求列表，统一管理所有请求监听。ImagePipelineFactory是实行单例模式，这个ImagePipelineFactory是在Fresco初始化时进行初始化的，它也是维持着一个ImagePipeline，就是说通过它一个维持一个请求的监听集合。这里我有点搞不明白，为什么一个image请求所有请求监听都要回调：

**[java]** view plain copy

print?

* @Override
* **public** **void** onRequestSuccess(ImageRequest request, String requestId, **boolean** isPrefetch) {
* **final** **int** numberOfListeners = mRequestListeners.size();
* **for** (**int** i = 0; i < numberOfListeners; ++i) {
* RequestListener listener = mRequestListeners.get(i);
* **try** {
* listener.onRequestSuccess(request, requestId, isPrefetch);
* } **catch** (Exception exception) {
* // Don't punish the other listeners if we're given a bad one.
* onException("InternalListener exception in onRequestSuccess", exception);
* }
* }
* }

现在可以明白为什么可以通过ImagePipeline获取到数据了吧。

说明那么多基本都通了，还有一点就是Q2问题，它是如何发送请求并获取的数据的呢，这大概是我们最想知道的吧。

我们看Q2 那个方法:

**[java]** view plain copy

print?

* //Returns a sequence that can be used for a request for a decoded image
* Producer<CloseableReference<CloseableImage>> producerSequence =
* mProducerSequenceFactory.getDecodedImageProducerSequence(
* imageRequest);//Q2

**[java]** view plain copy

print?

* /\*\*
* \* Returns a sequence that can be used for a request for a decoded image.
* \*
* \* @param imageRequest the request that will be submitted
* \* @return the sequence that should be used to process the request
* \*/
* **public** Producer<CloseableReference<CloseableImage>> getDecodedImageProducerSequence(
* ImageRequest imageRequest) {
* Producer<CloseableReference<CloseableImage>> pipelineSequence =
* getBasicDecodedImageSequence(imageRequest);
* **if** (imageRequest.getPostprocessor() != **null**) {
* **return** getPostprocessorSequence(pipelineSequence);//这是上个进程的数据
* } **else** {
* **return** pipelineSequence;
* }
* }

**[java]** view plain copy

print?

* **private** Producer<CloseableReference<CloseableImage>> getBasicDecodedImageSequence(
* ImageRequest imageRequest) {
* ...
* Uri uri = imageRequest.getSourceUri();
* **if** (UriUtil.isNetworkUri(uri)) {
* **return** getNetworkFetchSequence();
* } **else** **if** (UriUtil.isLocalFileUri(uri)) {
* **if** (MediaUtils.isVideo(MediaUtils.extractMime(uri.getPath()))) {
* **return** getLocalVideoFileFetchSequence();
* } **else** {
* **return** getLocalImageFileFetchSequence();
* }
* } **else** **if** (UriUtil.isLocalContentUri(uri)) {
* **return** getLocalContentUriFetchSequence();
* } **else** **if** (UriUtil.isLocalAssetUri(uri)) {
* **return** getLocalAssetFetchSequence();
* } **else** **if** (UriUtil.isLocalResourceUri(uri)) {
* **return** getLocalResourceFetchSequence();
* } **else** {
* **throw** **new** RuntimeException(
* "Unsupported image type! Uri is: " + uri.toString().substring(0, 30));
* }
* }

看到了么，这个可以获取很多的数据网络的、本地的、、我们关心的是获取网络数据，就是getNetworkFetchSequence()这个方法：

**[java]** view plain copy

print?

* /\*\*
* \* swallow result if prefetch -> bitmap cache get -> wait if scrolling ->
* \* background thread hand-off -> multiplex -> bitmap cache -> decode -> multiplex ->
* \* encoded cache -> disk cache -> (webp transcode) -> network fetch.
* \*/
* **private** **synchronized** Producer<CloseableReference<CloseableImage>> getNetworkFetchSequence() {
* **if** (mNetworkFetchSequence == **null**) {
* mNetworkFetchSequence =
* newBitmapCacheGetToDecodeSequence(getCommonNetworkFetchToEncodedMemorySequence());
* }
* **return** mNetworkFetchSequence;
* }

重点来了，我们知道我们请求并获取数据的过程是逆向的，就是提交请求-----判断有没有缓存-------没有缓存网络获取-----获取成功返回。Fresco巧妙的用了两个东西，producer和consumer，我们看官方解析：

**[java]** view plain copy

print?

* /\*\*
* \* Building block for image processing in the image pipeline.
* \*
* \* <p> Execution of image request consists of multiple different tasks such as network fetch,
* \* disk caching, memory caching, decoding, applying transformations etc. Producer<T> represents
* \* single task whose result is an instance of T. Breaking entire request into sequence of
* \* Producers allows us to construct different requests while reusing the same blocks.
* \*
* \* <p> Producer supports multiple values and streaming.
* \*
* \* @param <T>
* \*/
* **public** **interface** Producer<T>

每个image请求都有不同了流程任务，比如网络获取过程、硬盘缓存过程、内存缓存过程、编码过程、传输过程等，每个过程都通过一个producer标识，根据流程每完成一个producer任务就传递给下一个producer执行相应的任务，这样好处很明显，很清晰。执行任务通过producer进行，那么之间的数据传递怎么办，那就是consumer了。

**[java]** view plain copy

print?

* /\*\*
* \* Consumes data produced by {@link Producer}.<T>
* \*
* \* <p> The producer uses this interface to notify its client when new data is ready or an error
* \* occurs. Execution of the image request is structured as a sequence of Producers. Each one
* \* consumes data produced by producer preceding it in the sequence.
* \*
* \* <p>For example decode is a producer that consumes data produced by the disk cache get producer.
* \*
* \* <p> The consumer is passed new intermediate results via onNewResult(isLast = false) method. Each
* \* consumer should expect that one of the following methods will be called exactly once, as the very
* \* last producer call:
* \* <ul>
* \*   <li> onNewResult(isLast = true) if producer finishes successfully with a final result </li>
* \*   <li> onFailure if producer failed to produce a final result </li>
* \*   <li> onCancellation if producer was cancelled before a final result could be created </li>
* \* </ul>
* \*
* \* <p> Implementations of this interface must be thread safe, as callback methods might be called
* \* on different threads.
* \*
* \* @param <T>
* \*/
* **public** **interface** Consumer<T>

producer通过consumer将结果返回给client。

那么我们大概就可以猜测了它的流程类似就是producer1-----》producer12--------》producer3--------》producer4，它们之间传递consumer进行数据传递，按照这样的思路看看源码对不对。

前面我们看到了getNetworkFetchSequence()这个方法，这个方法就是获取网络数据的，

**[java]** view plain copy

print?

* /\*\*
* \* swallow result if prefetch -> bitmap cache get -> wait if scrolling ->
* \* background thread hand-off -> multiplex -> bitmap cache -> decode -> multiplex ->
* \* encoded cache -> disk cache -> (webp transcode) -> network fetch.
* \*/
* **private** **synchronized** Producer<CloseableReference<CloseableImage>> getNetworkFetchSequence() {
* **if** (mNetworkFetchSequence == **null**) {
* mNetworkFetchSequence =
* newBitmapCacheGetToDecodeSequence(getCommonNetworkFetchToEncodedMemorySequence());
* }
* **return** mNetworkFetchSequence;
* }

**[java]** view plain copy

print?

* **private** Producer<CloseableReference<CloseableImage>> newBitmapCacheGetToDecodeSequence(
* Producer<CloseableReference<PooledByteBuffer>><span style="color: rgb(255, 0, 0);"> nextProducer</span>) {
* DecodeProducer decodeProducer = mProducerFactory.newDecodeProducer(nextProducer);
* **return** newBitmapCacheGetToBitmapCacheSequence(decodeProducer);
* }

看到那个传递参数命名没有，叫nextproducer,而且 mProducerFactory.newDecodeProducer这个方法的作用就是创建了一个编码producer，getCommonNetworkFetchToEncodedMemorySequence()这个方法从名字来看就是获取网络数据后进行编码缓存的producer，所以这个方法 mProducerFactory.newDecodeProducer(nextProducer);可以理解为：解码的下一个步骤就是行进行网络或者并编码，同理newBitmapCacheGetToBitmapCacheSequence(decodeProducer)可以理解为：缓存bitmapcache下一个步骤就是进行解码。好像怪怪的，不过想想拿数据的大概就明白了，拿数据时是应该去缓存找，如果找到缓存就直接返回数据，如果没有缓存就进行下一步拿数据。所以上一个producer拿数据成功后就应该不会执行下一个producer，不成功就执行下一个produce去拿数据以此类推下去，应该是这样才对。

我们看下一个方法：

**[java]** view plain copy

print?

* /\*\*
* \* Bitmap cache get <pre name="code" class="java">/\*\*
* \* multiplex -> encoded cache -> disk cache -> (webp transcode) -> network fetch.
* \*/
* **private** **synchronized** Producer<CloseableReference<PooledByteBuffer>>
* getCommonNetworkFetchToEncodedMemorySequence() {
* **if** (mCommonNetworkFetchToEncodedMemorySequence == **null**) {
* mCommonNetworkFetchToEncodedMemorySequence =
* newEncodedCacheMultiplexToTranscodeSequence(mNetworkFetchProducer, /\* isLocal \*/**false**);
* **if** (mResizeAndRotateEnabledForNetwork) {
* mCommonNetworkFetchToEncodedMemorySequence =
* newResizeAndRotateImagesSequence(mCommonNetworkFetchToEncodedMemorySequence);
* }
* }
* **return** mCommonNetworkFetchToEncodedMemorySequence;
* }

这里的流程可以总结为：获取bitmap memory cache-------》启动ExecutorService进行后台处理-------》同一key多路请求合并成一个--------》获取解码了的bitmap memory cache

我们返回看getCommonNetworkFetchToEncodedMemorySequence()：

**[java]** view plain copy

print?

* /\*\*
* \* multiplex -> encoded cache -> disk cache -> (webp transcode) -> network fetch.
* \*/
* **private** **synchronized** Producer<CloseableReference<PooledByteBuffer>>
* getCommonNetworkFetchToEncodedMemorySequence() {
* **if** (mCommonNetworkFetchToEncodedMemorySequence == **null**) {
* mCommonNetworkFetchToEncodedMemorySequence =
* newEncodedCacheMultiplexToTranscodeSequence(mNetworkFetchProducer, /\* isLocal \*/**false**);
* **if** (mResizeAndRotateEnabledForNetwork) {
* mCommonNetworkFetchToEncodedMemorySequence =
* newResizeAndRotateImagesSequence(mCommonNetworkFetchToEncodedMemorySequence);
* }
* }
* **return** mCommonNetworkFetchToEncodedMemorySequence;
* }

**[java]** view plain copy

print?

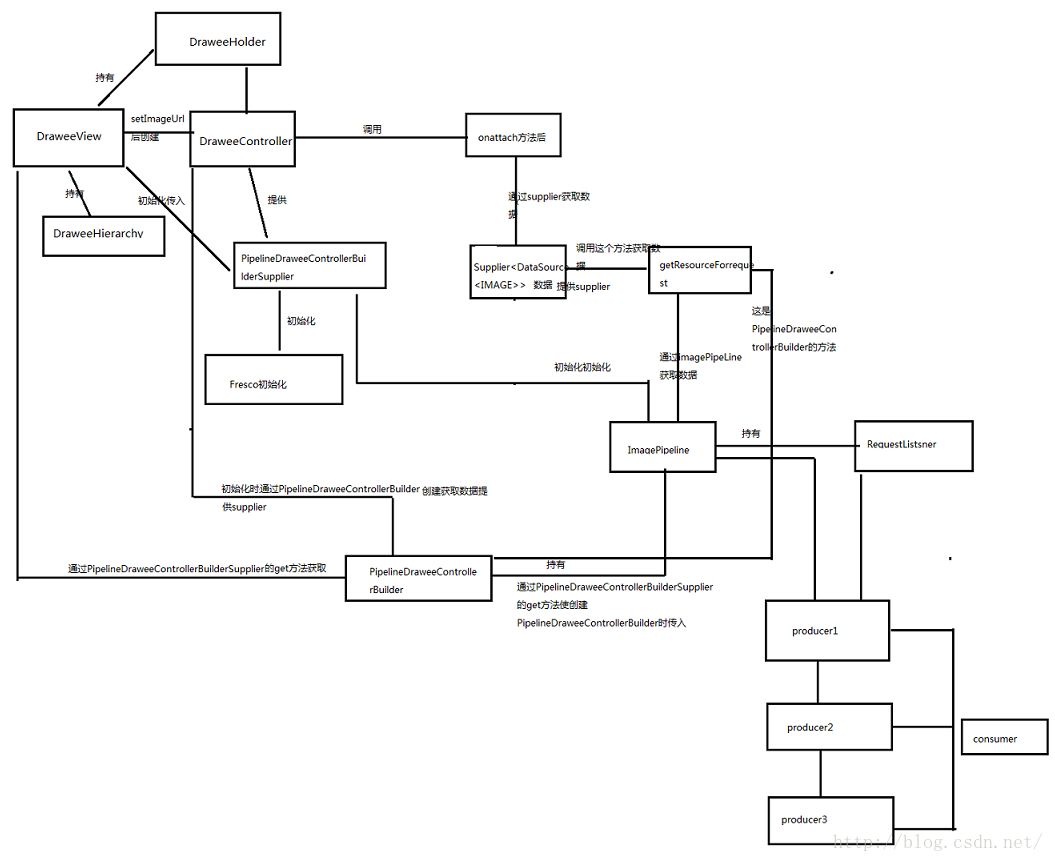
* /\*\*
* \* encoded cache multiplex -> encoded cache -> (disk cache) -> (webp transcode)
* \* @param nextProducer next producer in the sequence
* \* @param isLocal whether the image source is local or not
* \* @return encoded cache multiplex to webp transcode sequence
* \*/
* **private** Producer<CloseableReference<PooledByteBuffer>>
* newEncodedCacheMultiplexToTranscodeSequence(
* Producer<CloseableReference<PooledByteBuffer>> nextProducer,
* **boolean** isLocal) {
* **if** (Build.VERSION.SDK\_INT < Build.VERSION\_CODES.JELLY\_BEAN\_MR2) {
* nextProducer = mProducerFactory.newWebpTranscodeProducer(nextProducer);//这里进行网络获取数据
* }
* **if** (!isLocal) {
* nextProducer = mProducerFactory.newDiskCacheProducer(nextProducer);//如果需要缓存就硬盘就进行缓存
* }
* EncodedMemoryCacheProducer encodedMemoryCacheProducer =
* mProducerFactory.newEncodedMemoryCacheProducer(nextProducer);
* **return** mProducerFactory.newEncodedCacheKeyMultiplexProducer(encodedMemoryCacheProducer);
* }

其实最最简单的概括就是本文最上面的那张图了，事实上它又不止那么简单，这个详细说明也蛮多的，这里我只说个大概的思路，应该是没错的。那么还有它如何通过consumer进行数据传递，这也是蛮有意思，我想再下次具体详细看这个部分。

简单总结一下这个流程，Fresco是如何只使用setImageURI（）后自动完成了获取数据的：

DraweeView采用的 是典型的mvc模式，DraweeHierarchy 负责的是跟显示相关的，DraweeController 负责的是后台相关的，DraweeHolder主要是统筹两者的，Fresco的初始化方法执行后会初始化相关配置。以SimpleDraweeView为例，它的controller是PipelineDraweeController，Fresco在初始化时初始化一个PipelineDraweeControllerBuilderSupplier来提供这个controller，DraweeHolder主要通过controller启动后台服务获取数据，setController方法执行后后调用controller的onAttach()方法，这个方法会提交请求并获得数据，后将数据返回给ui线程，Fresco数据是封装成Supplier<DataSource<IMAGE>> 的形式返回，而真正将数据传递回来的是通过ImagePipeline，这个ImagePipeline会在PipelineDraweeControllerBuilderSupplier的构造器初始化创建一个，它会在Fresco初始化方法中进行初始化，SimpleDraweeView初始化方法中会调用PipelineDraweeControllerBuilderSupplier的get的方法将ImagePipeline返回，ImagePipeline有一个RequestListsner的成员变量，它管理这个一个监听列表，请求监听通过这个RequestListsner返回，而正确去获取数据的时候是启动一个后台服务进行数据获取的，整个数据获取有分多个阶段，Fresco通过producer划分这些阶段，比如缓存获取阶段、编码阶段、网络获取阶段，它会一个一个阶段去获取数据，如果producer1没有获取到数据就交给producer2进行获取数据以此类推下去，producer之间通过consumer返回数据，最终将数据返回到ui线程去。

画了个粗略的图：



### 12.[线程的生命周期](http://blog.csdn.net/mayouarebest8621/article/details/6755036)

**1.线程的生命周期**线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

(1)生命周期的五种状态

**新建（new Thread）**当创建Thread类的一个实例（对象）时，此线程进入新建状态（未被启动）。

例如：Thread  t1=new Thread();

**就绪（runnable）**线程已经被启动，正在等待被分配给CPU时间片，也就是说此时线程正在就绪队列中排队等候得到CPU资源。例如：**t1.start();**

**运行（running）**线程获得CPU资源正在执行任务（run()方法），此时除非此线程自动放弃CPU资源或者有优先级更高的线程进入，线程将一直运行到结束。

**死亡（dead）**

当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

自然终止：正常运行run()方法后终止

异常终止：调用**stop()**方法让一个线程终止运行

**堵塞（blocked）**

由于某种原因导致正在运行的线程让出CPU并暂停自己的执行，即进入堵塞状态。

正在睡眠：用sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。

正在等待：调用wait()方法。（调用motify()方法回到就绪状态）

被另一个线程所阻塞：调用suspend()方法。（调用resume()方法恢复）

**2.常用方法**

void run()   创建该类的子类时必须实现的方法

void start() 开启线程的方法

static void sleep(long t) 释放CPU的执行权，不释放锁

static void sleep(long millis,int nanos)

final void wait()释放CPU的执行权，释放锁

final void notify()

static void yied()可以对当前线程进行临时暂停（让线程将资源释放出来）

3.（1）结束线程原理：就是让run方法结束。而run方法中通常会定义循环结构，所以只要控制住循环即可

(2)方法----可以boolean标记的形式完成，只要在某一情况下将标记改变，让循环停止即可让线程结束

（3）public final void join()//让线程加入执行，执行某一线程join方法的线程会被冻结，等待某一线程执行结束，该线程才会恢复到可运行状态

4. **临界资源：多个线程间共享的数据称为临界资源**

（1）**互斥锁**

a.每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。

b.[**Java**](http://lib.csdn.net/base/java)对象默认是可以被多个线程共用的，只是在需要时才启动“互斥锁”机制，成为专用对象。

c.关键字synchronized用来与对象的互斥锁联系

d.当某个对象用synchronized修饰时，表明该对象已启动“互斥锁”机制，在任一时刻只能由一个线程访问，即使该线程出现堵塞，该对象的被锁定状态也不会解除，其他线程任不能访问该对象。

### 多个Activity同时启动同一个Activity，那Service有几个实例?

### 一个Activity startService之后，然后再bindService会出现什么情况？

### 抽象类可以有非抽象方法吗？

可以有，抽象类：包含抽象方法的类。只要是包含抽象方法就叫做抽象类，可以有方法体

### 冒泡排序

### 快速排序

### 字符串比较有几个方法

### asyncHttpClient的源码实现

### 线程保活的几种方式

**1) 将Service设置为前台服务**

**思路：**启用前台服务，主要是startForeground()

**保活程度：**一般情况下不被杀，部分定制ROM会在应用切到后台即杀

，会被 force stop 杀死

**代码实现：**

Notificationnotification = newNotification(R.drawable.queen2, "有消息来了"

, System.currentTimeMillis());

notification.setLatestEventInfo(this, "双11，上天猫！",

"一律5折", null);

//设置通知默认效果

notification.flags = Notification.FLAG\_SHOW\_LIGHTS;

startForeground(1, notification);

**2) 在service的onstart方法里返回 STATR\_STICK**

**思路：**其实就是onStartCommand中返回STATR\_STICK

**保活程度：**有次数和时间的限制

，会被 force stop 杀死

**代码实现：**

@Override

public int onStartCommand(Intent intent, int flags, int startId) {

// TODO Auto-generated method stub

return START\_STICKY;

//return super.onStartCommand(intent, flags, startId);

}

**3) 添加Manifest文件属性值为android:persistent=“true”**

**代码实现（清单文件中配置）：**

**保活程度：**一般情况下不被杀，会被 force stop 杀死

<application android:name="PhoneApp"

android:persistent="true"

android:label="@string/dialerIconLabel"

android:icon="@drawable/ic\_launcher\_phone">

注意：该方法需要系统签名

**4) 覆写Service的onDestroy方法**

**思路：**在onDestroy中再次启动该服务

**保活程度：**很弱，只在两种情况下work:正在运行里杀服务、DDMS里stop进程

**代码实现：**

@Override

public void onDestroy() {

Intent intent = new Intent(this, KeeLiveService.class);

startService(intent);

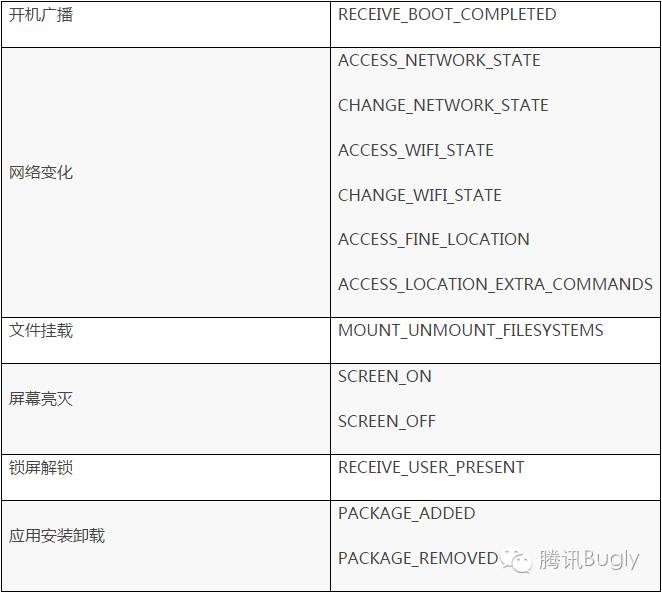
super.onDestroy();

}

**5) 监听一堆系统静态广播**

**思路：**在发生特定系统事件时，系统会发出响应的广播，通过在 AndroidManifest 中“静态”注册对应的广播监听器，即可在发生响应事件时拉活。

可以监听的系统静态广播列表如下：



**保活强度：**我们可以发现，这个方法都是监听系统的一些广播，所以我们需要在我们的应用中注册静态广播，但是静态广播又会出现问题，那就是在4.0版本以上，没有启动过的应用或Force-Stop后收不到静态广播，也就是说4.0以后，如果我们应用从未启动过，或者被Force-Stop杀死过，是无法接收到静态广播的。

如果是两个应用相互拉起，那么在一个应用内可发送带FLAG\_INCLUDE\_STOPPED\_PACKAGES的Intent，那即使另一个应用也是以上两种情况，也可以接收到系统的广播

应用1的代码实现：

//应用1，发送拉起服务的广播

Intent intent = new Intent();

intent.setAction("com.action.keepLive");

intent.addFlags(Intent.FLAG\_INCLUDE\_STOPPED\_PACKAGES);

this.sendBroadcast(intent);

应用2的代码实现：

<receiver android:name="com.yzy.supercleanmaster.receiver.KeepLiveReceiver">

<intent-filter>

<action android:name="com.action.keepLive" />

</intent-filter>

</receiver>

public class KeepLiveReceiver extends BroadcastReceiver{

//应用2中，接受应用1发送的广播，进行服务的拉起

@Override

public void onReceive(Context context, Intent intent) {

Intent i = new Intent(context, KeeLiveService.class);

context.startService(i);

}

}

**6) 监听第三方应用的静态广播**

**思路：**通过反编译第三方 Top 应用，如：手机QQ、[**微信**](http://lib.csdn.net/base/wechat)、支付宝、UC浏览器等，以及友盟、信鸽、个推等 SDK，找出它们外发的广播，在应用中进行监听，这样当这些应用发出广播时，就会将我们的应用拉活。

**保活强度：**

该方案的局限性除与系统广播一样的因素外，主要受如下因素限制：

1） 反编译分析过的第三方应用的多少

2） 第三方应用的广播属于应用私有，当前版本中有效的广播，在后续版本随时就可能被移除或被改为不外发，这些因素都影响了拉活的效果。

**7) AlarmManager唤醒**

**思路：**通过AlarmManager设置一个定时器，定时的唤醒服务

\*\*保活强度：\*\*killBackgroundProcess下，大部分情况work，

不敌force-stop，闹钟会被清除。

**代码实现：**

public void startKeepLiveService(Context context, int timeMillis,String action) {

//获取AlarmManager系统服务

AlarmManager alarmManager = (AlarmManager) context.getSystemService(Context.ALARM\_SERVICE);

//包装Intent

Intent intent = newIntent(context,KeepLiveServie.class);

intent.setAction(action);

PendingIntent pendingIntent = PendingIntent.getService(context,0,intent, PendingIntent.FLAG\_UPDATE\_CURRENT);

//添加到AlarmManager

alarmManager.setRepeating(AlarmManager.RTC\_WAKEUP,System.currentTimeMillis(),timeMillis,pendingIntent);

}

**8) 账户同步，定时唤醒**

\*\*思路：\*\*android系统里有一个账户系统，系统定期唤醒账号更新服务，同步的事件间隔是有限制的，最短1分钟。

**难点：**需要手动设置账户，你如何骗你的用户给你手动设置账户完了之后不卸载你，必须联网

**代码实现：**

① 建立数据同步系统（ContentProvider）

通过一个ContentProvider用来作数据同步，由于并没有实际数据同步，所以此处就直接建立一个空的ContentProvider即可

public class XXAccountProvider extends ContentProvider {

public static final String AUTHORITY = "包名.provider";

public static final String CONTENT\_URI\_BASE = "content://" + AUTHORITY;

public static final String TABLE\_NAME = "data";

public static final Uri CONTENT\_URI = Uri.parse(CONTENT\_URI\_BASE + "/" + TABLE\_NAME);

@Override

public boolean onCreate() {

return true;

}

@Nullable

@Override

public Cursor query(Uri uri, String[] projection, String selection,

String[] selectionArgs, String sortOrder) {

return null;

}

@Nullable

@Override

public String getType(Uri uri) {

return new String();

}

@Nullable

@Override

public Uri insert(Uri uri, ContentValues values) {

return null;

}

@Override

public int delete(Uri uri, String selection, String[] selectionArgs) {

return 0;

}

@Override

public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {

return 0;

}

}

然后再Manifest中声明

<provider

android:name="\*\*.XXAccountProvider"

android:authorities="@string/account\_auth\_provider"

android:exported="false"

android:syncable="true"/>

② 建立Sync系统 (SyncAdapter)

通过实现SyncAdapter这个系统服务后, 利用系统的定时器对程序数据ContentProvider进行更新，具体步骤为：

- 创建Sync服务

public class XXSyncService extends Service {

private static final Object sSyncAdapterLock = new Object();

private static XXSyncAdapter sSyncAdapter = null;

@Override

public void onCreate() {

synchronized (sSyncAdapterLock) {

if (sSyncAdapter == null) {

sSyncAdapter = new XXSyncAdapter(getApplicationContext(), true);

}

}

}

@Override

public IBinder onBind(Intent intent) {

return sSyncAdapter.getSyncAdapterBinder();

}

static class XXSyncAdapter extends AbstractThreadedSyncAdapter {

public XXSyncAdapter(Context context, boolean autoInitialize) {

super(context, autoInitialize);

}

@Override

public void onPerformSync(Account account, Bundle extras, String authority, ContentProviderClient provider, SyncResult syncResult) {

getContext().getContentResolver().notifyChange(XXAccountProvider.CONTENT\_URI, null, false);

}

}

}

声明Sync服务

<service

android:name="\*\*.XXSyncService"

android:exported="true"

android:process=":core">

<intent-filter>

<action

android:name="android.content.SyncAdapter"/>

</intent-filter>

<meta-data

android:name="android.content.SyncAdapter"

android:resource="@xml/sync\_adapter"/>

</service>

其中sync\_adapter为：

<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"

android:accountType="@string/account\_auth\_type"

android:allowParallelSyncs="false"

android:contentAuthority="@string/account\_auth\_provide"

android:isAlwaysSyncable="true"

android:supportsUploading="false"

android:userVisible="true"/>

参数说明：

android:contentAuthority 指定要同步的ContentProvider在其AndroidManifest.xml文件中有个android:authorities属性。

android:accountType 表示进行同步的账号的类型。

android:userVisible 设置是否在“设置”中显示

android:supportsUploading 设置是否必须notifyChange通知才能同步

android:allowParallelSyncs 是否支持多账号同时同步

android:isAlwaysSyncable 设置所有账号的isSyncable为1

android:syncAdapterSettingsAction 指定一个可以设置同步的activity的Action。

* 账户调用Sync服务
* 首先配置好Account（第三步），然后再通过ContentProvider实现
* 手动更新

public void triggerRefresh() {

Bundle b = new Bundle();

b.putBoolean(ContentResolver.SYNC\_EXTRAS\_MANUAL, true);

b.putBoolean(ContentResolver.SYNC\_EXTRAS\_EXPEDITED, true);

ContentResolver.requestSync(

account,

CONTENT\_AUTHORITY,

b);

}

添加账号

Account account = AccountService.GetAccount();

AccountManager accountManager = (AccountManager) context.getSystemService(Context.ACCOUNT\_SERVICE);

accountManager.addAccountExplicitly(...)

``

同步周期设置

ContentResolver.setIsSyncable(account, CONTENT\_AUTHORITY, 1);

ContentResolver.setSyncAutomatically(account, CONTENT\_AUTHORITY, true);

ContentResolver.addPeriodicSync(account, CONTENT\_AUTHORITY, new Bundle(), SYNC\_FREQUENCY);

“

③ 建立账号系统 (Account Authenticator)

通过建立Account账号，并关联SyncAdapter服务实现同步

- 创建Account服务

public class XXAuthService extends Service {

private XXAuthenticator mAuthenticator;

@Override

public void onCreate() {

mAuthenticator = new XXAuthenticator(this);

}

private XXAuthenticator getAuthenticator() {

if (mAuthenticator == null)

mAuthenticator = new XXAuthenticator(this);

return mAuthenticator;

}

@Override

public IBinder onBind(Intent intent) {

return getAuthenticator().getIBinder();

}

class XXAuthenticator extends AbstractAccountAuthenticator {

private final Context context;

private AccountManager accountManager;

public XXAuthenticator(Context context) {

super(context);

this.context = context;

accountManager = AccountManager.get(context);

}

@Override

public Bundle addAccount(AccountAuthenticatorResponse response, String accountType, String authTokenType, String[] requiredFeatures, Bundle options)

throws NetworkErrorException {

// 添加账号 示例代码

final Bundle bundle = new Bundle();

final Intent intent = new Intent(context, AuthActivity.class);

intent.putExtra(AccountManager.KEY\_ACCOUNT\_AUTHENTICATOR\_RESPONSE, response);

bundle.putParcelable(AccountManager.KEY\_INTENT, intent);

return bundle;

}

@Override

public Bundle getAuthToken(AccountAuthenticatorResponse response, Account account, String authTokenType, Bundle options)

throws NetworkErrorException {

// 认证 示例代码

String authToken = accountManager.peekAuthToken(account, getString(R.string.account\_token\_type));

//if not, might be expired, register again

if (TextUtils.isEmpty(authToken)) {

final String password = accountManager.getPassword(account);

if (password != null) {

//get new token

authToken = account.name + password;

}

}

//without password, need to sign again

final Bundle bundle = new Bundle();

if (!TextUtils.isEmpty(authToken)) {

bundle.putString(AccountManager.KEY\_ACCOUNT\_NAME, account.name);

bundle.putString(AccountManager.KEY\_ACCOUNT\_TYPE, account.type);

bundle.putString(AccountManager.KEY\_AUTHTOKEN, authToken);

return bundle;

}

//no account data at all, need to do a sign

final Intent intent = new Intent(context, AuthActivity.class);

intent.putExtra(AccountManager.KEY\_ACCOUNT\_AUTHENTICATOR\_RESPONSE, response);

intent.putExtra(AuthActivity.ARG\_ACCOUNT\_NAME, account.name);

bundle.putParcelable(AccountManager.KEY\_INTENT, intent);

return bundle;

}

@Override

public String getAuthTokenLabel(String authTokenType) {

// throw new UnsupportedOperationException();

return null;

}

@Override

public Bundle editProperties(AccountAuthenticatorResponse response, String accountType) {

return null;

}

@Override

public Bundle confirmCredentials(AccountAuthenticatorResponse response, Account account, Bundle options)

throws NetworkErrorException {

return null;

}

@Override

public Bundle updateCredentials(AccountAuthenticatorResponse response, Account account, String authTokenType, Bundle options)

throws NetworkErrorException {

return null;

}

@Override

public Bundle hasFeatures(AccountAuthenticatorResponse response, Account account, String[] features)

throws NetworkErrorException {

return null;

}

}

声明Account服务

<service

android:name="\*\*.XXAuthService"

android:exported="true"

android:process=":core">

<intent-filter>

<action

android:name="android.accounts.AccountAuthenticator"/>

</intent-filter>

<meta-data

android:name="android.accounts.AccountAuthenticator"

android:resource="@xml/authenticator"/>

</service>

其中authenticator为：

<?xml version="1.0" encoding="utf-8"?>

<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"

android:accountType="@string/account\_auth\_type"

android:icon="@drawable/icon"

android:smallIcon="@drawable/icon"

android:label="@string/app\_name"

/>

使用Account服务

* 同SyncAdapter，通过AccountManager使用
* - 申请Token主要是通过 [AccountManager.getAuthToken]系列方法
* - 添加账号则通过 [AccountManager.addAccount]
* - 查看是否存在账号通过 [AccountManager.getAccountsByType]

**保活强度：**

该方案适用于所有的 Android 版本，包括被 forestop 掉的进程也可以进行拉活。最新 Android 版本（Android N）中系统好像对账户同步这里做了变动，该方法不再有效。

### Android流量控制

### Android6.0瞌睡模式

Android6.0引入了下列特性来延长电池使用时间：

* App Standby（应用待机）。系统可以将未使用的应用置为AppStandby模式，临时限制这些应用的网络访问，延迟同步和后台job
* Doze（瞌睡）。如果用户在某个时间期限内未主动使用设备（屏幕关闭、设备静止），平台将进入深度睡眠状态。因为该特性要求平台检测静止状态，因此只有那些在Sensor HAL中实现了显式移动检测API的设备中才有效。
* Exemptions（豁免）。预装在设备上的系统应用和云消息服务，默认通常被豁免。应用开发者可以要求其应用使用这种设定。用户也可以在设置中来豁免某些应用。

注：Sensor HAL（HAL层） 介于 SensorManager（JNI层）和 内核驱动层 之间

### 同步锁的问题

**package** syncnized;

**class** SyncnizedDemo **implements** Runnable {

**private** **boolean** flag;

SyncnizedDemo(**boolean** flag) {

**this**.flag = flag;

}

**public** **void** run() {

**if** (flag) {

**while** (**true**)

**synchronized** (MyLock.*locka*) {

System.***out***.println(Thread.*currentThread*().getName() + "..if locka....");

**synchronized** (MyLock.*lockb*) {

System.***out***.println(Thread.*currentThread*().getName() + "..if lockb....");

}

}

} **else**

**while** (**true**) {

**synchronized** (MyLock.*lockb*) {

System.***out***.println(Thread.*currentThread*() + "..else lockb...");

**synchronized** (MyLock.*locka*) {

System.***out***.println(Thread.*currentThread*() + "..else locka...");

}

}

}

}

**static** **class** MyLock {

**public** **static** Object *locka* = **new** Object();

**public** **static** Object *lockb* = **new** Object();

}

**public** **static** **void** main(String[] args) {

SyncnizedDemo a = **new** SyncnizedDemo(**true**);

SyncnizedDemo b = **new** SyncnizedDemo(**false**);

Thread t1 = **new** Thread(a);

Thread t2 = **new** Thread(b);

t1.start();

t2.start();

}

}

### Handler HandlerThread区别

### HandlerThread的特点

* HandlerThread将loop转到子线程中处理，说白了就是将分担MainLooper的工作量，降低了主线程的压力，使主界面更流畅。
* 开启一个线程起到多个线程的作用。处理任务是串行执行，按消息发送顺序进行处理。HandlerThread本质是一个线程，在线程内部，代码是串行处理的。
* 但是由于每一个任务都将以队列的方式逐个被执行到，一旦队列中有某个任务执行时间过长，那么就会导致后续的任务都会被延迟处理。
* HandlerThread拥有自己的消息队列，它不会干扰或阻塞UI线程。
* 对于网络IO操作，HandlerThread并不适合，因为它只有一个线程，还得排队一个一个等着。

首先呢？HandlerThread面试的时候有的会问，但是面试官不直接问你是否知道HandlerThread以及用途和实现？面试官会问你：面试必问的一个题目：   
handler的消息机制等一系列问题，如果你说的还算可以，那么问题来了？   
接下来会问你假如在一个子线程（工作线程）中怎么使用handler？嘻嘻 ，如果你知道handler的消息机制，那么这个问题很好回答了，代码如下：

private final class WorkThread extends Thread {

private Handler mHandler;

public Handler getHandler() {

return mHandler;

}

public void quit() {

mHandler.getLooper().quit();

}

@Override

public void run() {

super.run();

//创建该线程对应的Looper,

// 内部实现

// 1。new Looper（）

// 2。将1步中的lopper 放在ThreadLocal里，ThreadLocal是保存数据的，主要应用场景是：线程间数据互不影响的情况

// 3。在1步中的Looper的构造函数中new MessageQueue();

//其实就是创建了该线程对用的Looper，Looper里创建MessageQueue来实现消息机制

//对消息机制不懂得同学可以查阅资料，网上很多也讲的很不错。

Looper.prepare();

mHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

Log.d("WorkThread", (Looper.getMainLooper() == Looper.myLooper()) + "," + msg.what);

}

};

//开启消息的死循环处理即：dispatchMessage

Looper.loop();

//注意这3个的顺序不能颠倒

Log.d("WorkThread", "end");

}

}

这里我们可以[**测试**](http://lib.csdn.net/base/softwaretest)下在另外一个子线程里通过WorkThread的getHandler给它发消息看下结果：

public void send(View view) {

new SendThread(mWorkThread.getHandler()).start();

}

private final class SendThread extends Thread {

private Handler mHandler;

SendThread(Handler handler) {

this.mHandler = handler;

}

@Override

public void run() {

super.run();

for (int i = 0; i < 3; i++) {

mHandler.sendEmptyMessage(0x1);

SystemClock.sleep(1000);

}

}

}

log:可以看到收到了消息并打印了是在非主线程

02-15 10:45:19.290 22660-22738/com.gxz.study D/WorkThread: false,1

02-15 10:45:20.290 22660-22738/com.gxz.study D/WorkThread: false,1

02-15 10:45:21.291 22660-22738/com.gxz.study D/WorkThread: false,1

这里有同学问 小伙子，你这里不对， Log.d(“WorkThread”, “end”);没打印。是的，Looper.loop();为阻塞函数，只有当调用mHandler.getLooper().quit()或者quitSafely()方法后，loop才会中止.那2个方法有毛区别呢？

1. 当我们调用Looper的quit方法时，实际上执行了MessageQueue中的removeAllMessagesLocked方法，该方法的作用是把MessageQueue消息池中所有的消息全部清空，无论是延迟消息（延迟消息是指通过sendMessageDelayed或通过postDelayed等方法发送的需要延迟执行的消息）还是非延迟消息。
2. 当我们调用Looper的quitSafely方法时，实际上执行了MessageQueue中的removeAllFutureMessagesLocked方法，通过名字就可以看出，该方法只会清空MessageQueue消息池中所有的延迟消息，并将消息池中所有的非延迟消息派发出去让Handler去处理，quitSafely相比于quit方法安全之处在于清空消息之前会派发所有的非延迟消息。
3. 无论是调用了quit方法还是quitSafely方法只会，Looper就不再接收新的消息。即在调用了Looper的quit或quitSafely方法之后，消息循环就终结了，这时候再通过Handler调用sendMessage或post等方法发送消息时均返回false，表示消息没有成功放入消息队列MessageQueue中，因为消息队列已经退出了。
4. 需要注意的是Looper的quit方法从API Level 1就存在了，但是Looper的quitSafely方法从API Level 18才添加进来。

HandlerThread的用法

前面扯远了，但是你必须知道的东西。那就有同学问了好麻烦啊上面的，假入我以后在工作中用的话我还得注意 Looper.prepare(); new mHandler(); Looper.loop();的顺序。呵呵。。。   
哈哈，同学，这个HandlerThread就是解决这个的。google工程师早知道你会这么问。   
我们先看怎么用，我们把上面的测试例子用HandlerThread实现然后在分析内部原理。

//1.初始化,参数为名字，也就是线程的名字，后面我们会结合源码来看

mHandlerThread = new HandlerThread("WorkThread");

//必须调用start方法，因为HandlerThread继承自Thread来启动线程

mHandlerThread.start();

//初始化Handler,只是传递了一个mHandlerThread内部的一个looper

mHandler = new Handler(mHandlerThread.getLooper()) {

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

Log.d("WorkThread", (Looper.getMainLooper() == Looper.myLooper()) + "," + msg.what);

}

};

//2.使用

public void send(View view) {

new SendThread(mHandler).start();

}

private final class SendThread extends Thread {

private Handler mHandler;

SendThread(Handler handler) {

this.mHandler = handler;

}

@Override

public void run() {

super.run();

for (int i = 0; i < 3; i++) {

mHandler.sendEmptyMessage(0x1);

SystemClock.sleep(1000);

}

}

}

//3.别忘记释放，停止Looper消息循环，内部还是调用了looper的quit,及时释放防止内存泄漏哦！！

mHandlerThread.quit();

结果（一毛一样哦）：

02-15 11:11:24.738 13536-13787/com.gxz.study D/WorkThread: false,1

02-15 11:11:25.739 13536-13787/com.gxz.study D/WorkThread: false,1

02-15 11:11:26.740 13536-13787/com.gxz.study D/WorkThread: false,1

因此

1. 如果我们有耗时的任务处理可以通过HandlerThread获取looper,looper进而构造Handler，然后通过Handler的post(Runnable r)在handleMessage里进行处理耗时处理。
2. 很方便进行线程间的通信。

## HandlerThread的源码分析

* 原理：HandlerThread其实是extends Thread，内部的run方法调用了Looper.prepare()方法和 Looper.loop()；哈哈，和我们之前写的第一种方法一样。

public class HandlerThread extends Thread {

int mPriority;//线程优先级

int mTid = -1;//线程id

Looper mLooper;//Looper对象

public HandlerThread(String name) {

super(name);

mPriority = Process.THREAD\_PRIORITY\_DEFAULT;

}

* 对于线程，我们还是主要看run方法，源码很短，直接全部贴了。我们清楚的看到了Looper.prepare和Looper.loop的创建，这里和最开始说的一样。

@Override

public void run() {

mTid = Process.myTid();

Looper.prepare();

//后面会分析

synchronized (this) {

mLooper = Looper.myLooper();

notifyAll();

}

Process.setThreadPriority(mPriority);

onLooperPrepared();

Looper.loop();

mTid = -1;

}

但是这多了一个onLooperPrepared()方法，这里可以根据名字看出来，当Looper.prepare()创建完后，我们可以做一些初始化的东西，这当然是在子线程里，我们看下这个方法的实现。空实现，如果你要做些初始化的准备工作可以extends HandlerThread重写onLooperPrepared方法即可。

/\*\*

\* Call back method that can be explicitly overridden if needed to execute some

\* setup before Looper loops.

\*/

protected void onLooperPrepared() {

}

* 退出/结束方法：quit()方法或quitSafely()方法。实际上是调用了looper.quit方法和looper.quitSafely方法，上文中的区别也说到了。

public boolean quit() {

Looper looper = getLooper();

if (looper != null) {

looper.quit();

return true;

}

return false;

}

public boolean quitSafely() {

Looper looper = getLooper();

if (looper != null) {

looper.quitSafely();

return true;

}

return false;

}

* 最后还有一点，就是在上面的例子中我们构造Handler的时候传入了Looper,通过HandlerThread的getLooper()，源码如下。这里如果线程没有start活着一些其它原因，该线程没有处于一种存活的状态会返回null，

/\*\*

\* This method returns the Looper associated with this thread. If this thread not been started

\* or for any reason is isAlive() returns false, this method will return null. If this thread

\* has been started, this method will block until the looper has been initialized.

\* @return The looper.

\*/

public Looper getLooper() {

if (!isAlive()) {

return null;

}

// If the thread has been started, wait until the looper has been created.

synchronized (this) {

while (isAlive() && mLooper == null) {

try {

wait();

} catch (InterruptedException e) {

}

}

}

return mLooper;

}

该函数有可能阻塞，目的是为了防止获取的looper对象为空，使用了wait方法，并且使用了局部synchronized，锁住了当前的对象，那什么时候唤醒等待呢？当然是初始化完该线程关联looper对象的地方，也就是run方法。也就是你构造完HandlerThread必须调用start方法。对于synchronized和notifyAll和wait一些线程同步处理的操作，可以参考一些资料。

synchronized (this) {

mLooper = Looper.myLooper();

notifyAll();

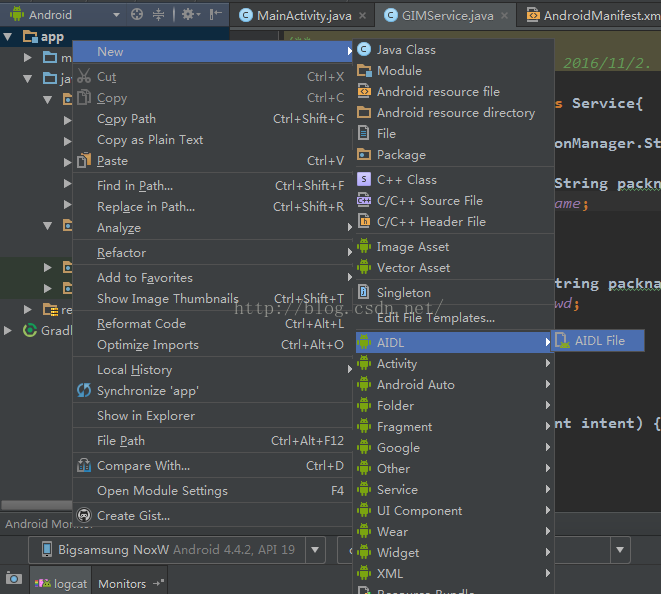
}

## 结束

这里我们学习了HandlerThread类的简单用法以及内部源码以及实现原理。   
同时也介绍了一些面试必问handler的消息机制。so，当再有面试官问你文章最开始的在一个子线程（工作线程）中怎么使用handler的问题，你可以说下怎么使用，同时最后说一个[**Android**](http://lib.csdn.net/base/android)提供的类-HandlerThread也很不错哦！！

### AIDL的使用

1.在需要分享出信息的软件里面创建AIDL文件。



之后命令生成一个XXXX.AIDL文件

删掉里面自带的方法。添加自己需要的方法

我添加了两个String的方法 一个用来获取name 一个用来获取pwd

**[java]** view plain copy

print?

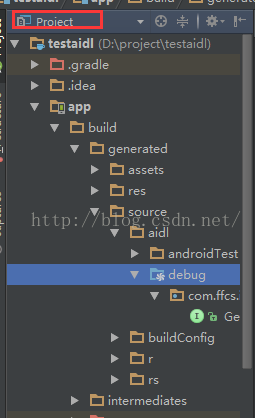
1. **interface** GetInformationManager {
3. String getName(String packname);
5. String getPwd(String packname);
6. }

我里面要求传参数 是为了做一些判断而已，用来判断允许哪些第三方软件可以访问我的数据

这之后请clean或者rebuild你的项目，为了让AS生成相对应的[**Java**](http://lib.csdn.net/base/java)文件

没错之后再进行下一步。

生成的位置如下



2.创建一个自己的服务来实现AIDL的方法

代码：

**[java]** view plain copy

print?

1. /\*\*
2. \* Created by Administrator on 2016/11/2.
3. \*/
4. **public** **class** GIMService **extends** Service{
6. **private** **final** GetInformationManager.Stub manager = **new** GetInformationManager.Stub() {
7. @Override
8. **public** String getName(String packname) **throws** RemoteException {
9. **return** Tools.username;
10. }
12. @Override
13. **public** String getPwd(String packname) **throws** RemoteException {
14. **return** Tools.userpwd;
15. }
16. };
18. @Nullable
19. @Override
20. **public** IBinder onBind(Intent intent) {
22. **return** manager;
23. }
24. }

由于测试，我就没在get的两个方法里面添加判定。

3.在AndroidManifest.xml文件里面添加服务即可

代码：

**[java]** view plain copy

print?

1. <service
2. android:name="com.ffcs.aidl.GIMService"
3. android:enabled="true"
4. android:exported="true">
5. <intent-filter>
6. <action android:name="android.intent.action.GIMService" />
7. </intent-filter>
8. </service>

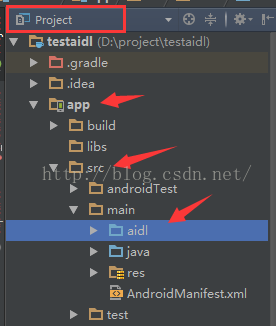
至此，自己的软件 设置算是完全完成了。

在下面是 第三方软件需要加入的东西设置了。

1.将自己软件里面新建出来的xxx.AIDL文件拷贝出来，让第三方软件放进自己的软件。

拷贝的话建议是把整个aidl文件夹拷贝到第三方软件内。

放的位置：

 拷贝完成后还是需要clean或者rebuild你的项目，为了让AS生成相对应的JAVA文件

没错后就可以进行下一步了。 2.在需要调用的地方开启服务

设置了一个新的test项目在主界面加入了一个按钮

布局图如下



代码如下

**[java]** view plain copy

print?

1. **public** **class** MainActivity **extends** AppCompatActivity {
3. @Override
4. **protected** **void** onCreate(Bundle savedInstanceState) {
5. **super**.onCreate(savedInstanceState);
6. setContentView(R.layout.activity\_main);
8. Button button = (Button) findViewById(R.id.button);
9. button.setOnClickListener(**new** View.OnClickListener() {
10. @Override
11. **public** **void** onClick(View v) {
12. **if** (!mBound){
13. startService();
14. Toast.makeText(MainActivity.**this**,"服务正在重连",Toast.LENGTH\_SHORT).show();
15. **return**;
16. }
17. **if** (manager == **null**) **return**;
18. **try** {
19. String name = manager.getName("");
20. String pwd = manager.getPwd("");
21. Log.e("账号",name);
22. Log.e("密码",pwd);
23. } **catch** (RemoteException e) {
24. e.printStackTrace();
25. }
26. }
27. });
28. }
30. **private** **void** startService(){
31. Intent intent = **new** Intent();
32. intent.setAction("android.intent.action.GIMService");
33. intent.setPackage("com.ffcs.stb");
34. bindService(intent,mServiceConnection, Context.BIND\_AUTO\_CREATE);
35. }
37. @Override
38. **protected** **void** onStart() {
39. **super**.onStart();
40. **if** (!mBound){
41. startService();
42. }
43. }
45. @Override
46. **protected** **void** onStop() {
47. **super**.onStop();
48. **if** (mBound){
49. unbindService(mServiceConnection);
50. mBound = **false**;
51. }
52. }
54. **private** GetInformationManager manager = **null**;
55. //默认未连接服务
56. **private** **boolean** mBound = **false**;
58. **private** ServiceConnection mServiceConnection = **new** ServiceConnection() {
59. @Override
60. **public** **void** onServiceConnected(ComponentName name, IBinder service) {
61. manager = GetInformationManager.Stub.asInterface(service);
62. mBound = **true**;
63. }
65. @Override
66. **public** **void** onServiceDisconnected(ComponentName name) {
67. mBound = **false**;
68. }
69. };
70. }

至此第三方的项目就可以获取数据了。

运行结果如下：



最后说下注意点吧。

1.    5.0之后隐式Intent 需要设置活动和包名，不然会错误

2.    创建和拷贝.AIDL文件之后需要重建下项目  不然不会生成相应的JAVA文件，导致写服务一直点不出来

3.    还有就是  需要你软件开启之后，第三方软件才可以获取服务从而获取数据

最后是，我直接用项目写的，源码可能就没办法发了！

### 301 302 404 200 500

|  |
| --- |
| 200（成功）  服务器已成功处理了请求。通常，这表示服务器提供了请求的网页。如果是对您的 robots.txt 文件显示此状态码，则表示 Googlebot 已成功检索到该文件。 |
| [301](http://blog.liujason.com/tag/301)（永久移动）  请求的网页已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。您应使用此代码告诉 Googlebot 某个网页或网站已永久移动到新位置。 |
| [302](http://blog.liujason.com/tag/302)（临时移动）  服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来响应以后的请求。此代码与响应 GET 和 HEAD 请求的 [301](http://blog.liujason.com/tag/301) 代码类似，会自动将请求者转到不同的位置，但您不应使用此代码来告诉 Googlebot 某个网页或网站已经移动，因为 Googlebot 会继续抓取原有位置并编制索引。  404（未找到）  服务器找不到请求的网页。例如，对于服务器上不存在的网页经常会返回此代码。  如果您的网站上没有 robots.txt 文件，而您在 Google 网站管理员工具“诊断”标签的 robots.txt 页上看到此状态码，则这是正确的状态码。但是，如果您有 robots.txt 文件而又看到此状态码，则说明您的 robots.txt 文件可能命名错误或位于错误的位置（该文件应当位于顶级域，名为 robots.txt）。  如果对于 Googlebot 抓取的网址看到此状态码（在”诊断”标签的 HTTP 错误页面上），则表示 Googlebot 跟随的可能是另一个页面的无效链接（是旧链接或输入有误的链接）。  500（服务器内部错误）  服务器遇到错误，无法完成请求。 Collections的组织架构图什么叫Serialization  Serialization是指把类或者基本的数据类型持久化(persistence)到数据流(Stream)中，包括文件、字节流、网络数据流。           JAVA中实现serialization主要靠两个类：ObjectOuputStream和ObjectInputStream。他们是JAVA IO系统里的OutputStream和InputStream的子类。 双锁机制 **一、饿汉模式**  饿汉模式代码     1. /\*\* 2. \* @ClassName: Singleton\_Simple 3. \* @Description: 单例模式——饿汉模式 4. \* @author Ran 5. \* @date 2011-2-4 上午12:46:15 6. \* 7. \*/ 8. public class Singleton\_Simple { 10. private static final Singleton\_Simple simple = new Singleton\_Simple(); 12. private Singleton\_Simple(){} 14. public static Singleton\_Simple getInstance(){ 15. return simple; 16. } 18. }       说明：顾名思义，饿汉模式就是在jvm进程启动并在我们主动使用该类的时候就会在内存中初始化一个单例对象，当我们调用getInstance()的时候直接获取该对象，他的创建是在我们调用getInstance()静态方法之前！    **二、懒汉模式**  懒汉模式代码     1. /\*\* 2. \* @ClassName: Singleton\_lazy 3. \* @Description: 单例模式——懒汉模式 4. \* @author Ran 5. \* @date 2011-2-4 上午12:48:41 6. \* 7. \*/ 8. public class Singleton\_lazy { 10. private static Singleton\_lazy lazy = null; 12. private Singleton\_lazy(){} 14. public static synchronized Singleton\_lazy getInstance(){ 15. if( lazy == null ){ 16. lazy = new Singleton\_lazy(); 17. } 18. return lazy; 19. } 20. }       说明：懒汉模式是相对于饿汉模式而言的，在jvm进程启动并在我们主动使用该类的时候不会在内存中初始化一个单例对象，只有当我们调用getInstance()的时候才去创建该对象，他的创建是在我们调用getInstance()静态方法之后，为了并没现象同步问题，我们在getInstance()方法上加了一个锁，这个方法每次只允许一个线程进来，虽然同步问题是解决了，但是相应的性能问题就出现了。    **三、双锁机制**    双锁机制 代码     1. /\*\* 2. \* @ClassName: Singleton\_DoubleKey 3. \* @Description: 单例模式——双锁机制 4. \* @author Ran 5. \* @date 2011-2-4 上午12:53:50 6. \* 7. \*/ 8. public class Singleton\_DoubleKey { 10. private static Singleton\_DoubleKey doubleKey = null; 12. private Singleton\_DoubleKey (){} 14. public static Singleton\_DoubleKey getInstance(){ 15. if( doubleKey == null ){    //① 16. synchronized(Singleton\_DoubleKey.class){    //② 17. if( doubleKey == null ){    //③ 18. doubleKey = new Singleton\_DoubleKey();  //④ 19. } 20. } 21. } 22. return doubleKey; 23. } 24. }        说明：双锁机制的出现是为了解决前面同步问题和性能问题，看上面的代码，简单分析下确实是解决了多线程并行进来不会出现重复new对象，而且也实现了懒加载，但是当我们静下来并结合java虚拟机的类加载过程我们就会发现问题出来了，对于JVM加载类过程不熟悉的，这里我简单介绍下，熟悉的跳过这段(当然，既然你熟悉就自然会知道双锁的弊端了)。  jvm加载一个类大体分为三个步骤：   1. 加载阶段：就是在硬盘上寻找java文件对应的class文件，并将class文件中的二进制数据加载到内存中，将其放在运行期数据区的方法区中去，然后在堆区创建一个java.lang.Class对象，用来封装在方法区内的数据结构； 2. 连接阶段：这个阶段分为三个步骤，步骤一：验证，验证什么呢？当然是验证这个class文件里面的二进制数据是否符合java规范咯；步骤二：准备，为该类的静态变量分配内存空间，并将变量赋一个默认值，比如int的默认值为0；步骤三：解析，这个阶段就不好解释了，将符号引用转化为直接引用，涉及到指针，这里不做多的解释； 3. 初始化阶段：当我们主动调用该类的时候，将该类的变量赋于正确的值(这里不要和第二阶段的准备混淆了)，举个例子说明下两个区别，比如一个类里有private static int i = 5; 这个静态变量在"准备"阶段会被分配一个内存空间并且被赋予一个默认值0，当道到初始化阶段的时候会将这个变量赋予正确的值即5，了解了吧！   好了，上面大体介绍了jvm类加载过程，回到我们的双锁机制上来分析下问题出在了哪里？假如有两个并发线程a、b，a线程主动调用了静态方法getInstance()，这时开始加载和初始化该类的静态变量，b线程调用getInstance()并等待获得同步锁，当a线程初始化对象过程中，到了第二阶段即连接阶段的准备步骤时，静态变量doubleKey 被赋予了一个默认值，但是这时还没有进行初始化，这时当a线程释放锁后，b线程判断doubleKey ！= null，则直接返回了一个没有初始化的doubleKey 对象，问题就出现在这里了，b线程拿到的是一个被赋予了默认值但是未初始化的对象，刚刚可以通过锁的检索！    所以对于上面的几个模式还是推荐使用第一种，在jvm加载类的时候就初始化一个对象，也避免了同步问题。 |